Structure-Adaptable Digital Neural Networks

THÈSE Nº 2052

Présentée au Département d'informatique
ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE
pour l'obtention du grade de Docteur ès sciences techniques

par

Andrés Pérez-Uribe
ingeniero electricista,
Universidad del Valle, Cali, Colombie
de nationalité colombienne

presentée au jury:

Prof. D. Mange	directeur de thèse
Prof. I. Aleksander	rapporteur
Prof. J. Cabestany	rapporteur
Prof. W. Gerstner	rapporteur
Dr. P. Marchal	rapporteur
Prof. E. Sanchez	rapporteur

Lausanne, EPFL
1999

Abstract

Neuroengineers have came up with the idea of *artificial neural networks*, massively parallel computing machines inspired by biological nervous systems. Such systems offer a new paradigm, where *learning from examples* or *learning from interaction* replaces programming. They are basically composed of interconnected simple processing elements (i.e., artificial neurons or simply neurons).

A predominant approach in the field of artificial neural networks consists of using a database to train the system by applying a so-called *learning algorithm* to modify the interconnection dynamics between artificial neurons, using a predetermined network topology (i.e., the number of artificial neurons and the way they are interconnected). When the training process ends, the system remains fixed, and can be considered a *learned system*.

However, in applications where the entire training database is not available or may change in time, the learning process must persist, giving rise to *learning systems*. A problem of such systems is how to preserve what has been previously learned while continuing to incorporate new knowledge. Learning systems overcome such problems by a local adaptation process and by offering the possibility of dynamically modifying the network's topology.

Most artificial neural network applications today are executed as conventional software simulations on sequential machines with no dedicated hardware. *Reconfigurable hardware* can provide the best features of both dedicated hardware circuits and software implementations: high-density and high-performance designs, and rapid prototyping and flexibility.

The main goal in this thesis was therefore the development of structure-adaptable digital neural networks with continual learning capabilities, using field-programmable logic devices. Neural network models with modifiable structure have already been developed, but are usually computationally intensive. Therefore, we have developed the *FAST* neural architecture, an unsupervised learning with Flexible Adaptable-Size Topology that does not require intensive computation to learn and reconfigure its structure, and can thus exist as a stand-alone learning machine that can be used, for example, in real-time control applications, such as robot navigation.

The FAST learning system was conceived to handle the problem of *dynamic categorization* or *online clustering*. In this thesis, we are interested in two different categorization tasks: in the first task, related to image processing, the process of color categorization is used for image segmentation, while in the second task, related to neurocontrol, an autonomous "intelligent" system self-categorizes (or clusters) its sensor readings in order to integrate the relentless bombardment of signals coming from the

environment (i.e., its sensations). In this second system, a different learning process had to be considered to allow the system to generate behavioral responses as a function of its sensations. Indeed, other types of learning, such as *reinforcement learning*, seem to be essential to learn through interactions with the environment.

We have used this latter paradigm to learn game strategies and solve Markovian and non-Markovian maze tasks. Finally, we combined the capabilities of the FAST neural architecture with reinforcement learning techniques and developed a *neurocontroller* architecture, which we tested with the inverted pendulum problem, and then used in a navigation learning task with an autonomous mobile robot. Finally, we devised a digital implementation of these neurocontrollers and developed an FPGA board to control an off-the-shelf miniaturized mobile robot.

Résumé

Les réseaux neuromimétiques, outils de calcul massivement parallèles inspirés par l'étude du système nerveux, exploitent un nouveau paradigme dans lequel l'apprentissage, par l'exemple ou par interaction, remplace la programmation. Ils sont essentiellement composés d'unités de traitement élémentaires (ou neurones) interconnectées selon une certaine topologie.

L'approche prédominante dans le domaine des réseaux de neurones artificiels consiste à définir une topologie, puis à utiliser une base de données pour entraîner le système en exploitant un *algorithme d'apprentissage* permettant une modification de la dynamique des interconnexions. Une fois le processus d'entraînement achevé, le réseau est figé et donc peut être consideré comme un *système appris*.

Cette approche ne convient cependant pas à des applications où la base d'entraînement n'est que partiellement connue ou susceptible d'évoluer au cours du temps. Dans de telles situations, le réseau doit pouvoir apprendre à tout moment de nouvelles données, tout en préservant les connaissances acquises. Les *systèmes à apprentissage continu* résolvent ce problème. Ils subissent une adaptation locale et modifient dynamiquement leur topologie.

La majorité des systèmes neuromimétiques est implantée en logiciel dans des architectures mono (ou multi) processeur(s) ne possédant aucun matériel dédié à ce type de problème. Le *matériel reconfigurable* combine les avantages des circuits dédiés (ASIC) et du logiciel : prototypage rapide, flexibilité ou obtention de circuits à hautes performances.

La conception de réseaux neuromimétiques à structure adaptable dotés de capacités d'apprentissage constitue l'objectif principal de cette thèse. Par le passé, des chercheurs ont déjà développés pareils systèmes. Cependant, leurs modèles requièrent une charge de calcul considérable et ne s'implantent que difficilement en matériel. Pour remédier à ce problème, nous proposons l'architecture *FAST* (Flexible Adaptable-Size Topology).

Le réseau FAST permet le traitement des problèmes de discrimination dynamique et de *clustering* en ligne. Dans cette thèse, nous sommes particulièrement intéressés par deux tâches différentes : la première consiste à segmenter des images à l'aide du processus de discrimination de couleurs. La seconde est liée au domaine du neurocontrôle. Nous exploitons le principe de *clustering* pour permettre à un système autonome "intelligent" d'établir des relations entre les différentes activations de ses capteurs (sensations ou stimuli provenant de son environnement). Un mécanisme d'apprentissage par renforcement permet ensuite l'association d'un comportement particulier à chacune des sensations.

Nous avons utilisé l'apprentissage renforcé pour la recherche de stratégies dans le domaine des jeux et pour la navigation dans des labyrinthes markoviens et non markoviens. Nous avons ensuite couplé l'architecture FAST à des systèmes exploitant l'apprentissage renforcé afin d'obtenir des *neurocontrôleurs* résolvant les problèmes du pendule inversé et de la navigation d'un robot mobile autonome. Finalement, nous avons implanté un neurocontrôleur à l'aide de circuits programmables et développé une carte à FPGA pour le pilotage d'un robot mobile disponible dans le commerce.

Acknowledgments

To all my teachers...

Although, from a probabilistic point of view, the text of this thesis *could* be obtained by pure chance (by an infinite number of monkeys, for example), such an approach might well haver required millions of years. Instead, this thesis is simply the culmination of a few years of learning, and it is not by chance that it deals with learning. This learning process as been possible thanks to many people that believed in me and kept on teaching me many things about life, about science and engineering, about learning. I am profoundly grateful to all my teachers and it is therefore only fitting that I dedicate this thesis to them.

First of all, I am immensely grateful to my parents Bernardo and Alicia, and to my family, particularly to my aunt Esperanza. They have always supported me and taught me the things that most matter in life. Moreover, I am very grateful to all of those that motivated and encouraged me to pursue a life in academia.

I am very grateful to my brother for his music, which helped me concentrate, relax, and even have new ideas. I also thank very much my sister for introducing me to very interesting ideas in the field of psychology.

I thank Beatriz Helena, who supported me always, partook of my achievements, and encouraged me when things did not go so well. We have been together, during the past few years, helping each other discover and learn about life. I am very grateful to her.

I thank Eduardo (Sanchez), a good friend and a mentor. He and his family welcomed me into their house when I came to Switzerland and taught me much about living in this country. I am very grateful to them. Eduardo is not by chance one of the experts for this thesis: he was always by my side and ready to talk and to help me find ways to overcome problems in my research. I thank him very much for taking time to give me advice on many occasions and to talk about many other interesting scientific topics.

I thank Professor Daniel Mange for receiving me in his great laboratory and for accepting to direct this thesis. I cannot describe how fascinating my experience was at the LSL. The things I have learned during the past few years have changed my life. Moreover, I have learned many things from Daniel, such as order, pragmatism, and all those small details that bring success to a research group. Finally, I thank him for entrusting me with very challenging tasks.

I am very grateful to Marlyse Taric, our lab's secretary, for all her help and her friendliness.

I am grateful to Pierre Marchal, who, beyond representing the *Centre Suisse d'Electronique et de Microtecnique* (CSEM), the research center that supported this research,

was very interested in my research and was present throughout its development. I thank him for his contribution and friendship.

My profound acknowledgments to the experts of this thesis, Prof. I. Aleksander, Prof. J. Cabestany, Prof. W. Gerstner, Dr. P. Marchal, and Prof. E. Sanchez, and Prof. J.-D. Nicoud, president of the jury, for their careful reading and useful suggestions.

I thank all members of the laboratory for making my stay very pleasant and fruitful. I am particularly grateful to Moshe Sipper for his help during the first stages of my thesis, for our fruitful discussions, and for his advice. I am profoundly grateful to Gianluca Tempesti, Jean-Luc Beuchat, and Hector Fabio Restrepo for their proofreading of this thesis, their comments, and their suggestions. I thank very much André Badertscher for his friendliness, for sharing his remarkable pictures of animals and landscapes, and for his practical advice on real-world problems. His outstanding technical skills and those of Georges Vaucher from the ACORT were essential for the implementation of the robot's neurocontroller. I thank Carlos Andrés Peña for our useful discussions about computational intelligence and many other subjects. I thank Hector Fabio Restrepo for his unconditional help. I thank Jean-Michel Puiatti particularly for his friendship and help. I am also grateful to Dominik Madon, Jacques-Olivier Haenni, and Christian Iseli for their competent administration of our system and for all their help. I thank Emeka Mosanya, Serge Durand, and Flavio Rampogna for their help on practical issues related to FPGA devices. I thank Jean-Luc Beuchat for sharing his love for mountains, glaciers, waterfalls, and Latex and xfig. I thank Prof. Marco Tomassini for entrusting me with very challenging jobs. I thank Mathieu Capcarrere for sharing his love for philosophy and for taxing my French.

I am very grateful to all those friends who were like a family throughout the last few years. Some of them living in Switzerland, others keeping in touch from Colombia, Spain, and the USA. I thank them not only for their friendship and camaraderie but also for our interesting discussions about engineering, science, politics, religion, economy, and education. Our daily discussions at dinner time, during the first years at the EPFL, were particularly enriching. Finally, I thank them for sharing very pleasant moments in the mountains, and exquisite gastronomic experiences.

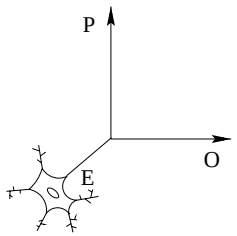
I am also grateful to Jaime Velasco, who introduced me to the world of digital design and FPGAs, to Gabriel J. Gomez for introducing me to the wonderful world of artificial neural networks, and to Gonzalo Ulloa and Jerry de Raad who convinced me to come to Switzerland and with whom I started my "professional" life.

I thank Christof Teuscher and Yuri López de Meneses for their helpful suggestions on the implementation of the FPGA board, and Dr. Paolo Ienne, Dr. Dario Floreano, Dr. José del R. Millan, Prof. Richard Sutton, Prof. Charles Anderson, Dr. Cristopher Molina, Prof. Henrik Lund, and Prof. Domingo Benitez, for their helpful comments throughout my research. I thank the *Commission fédérale des bourses pour étudiants étrangers* for its financial support during my first two years in Switzerland, and the *Centre Swiss d'Electronique et de Microtechnique* (CSEM) and the EPFL for supporting me during the last three years. There are far too many more people to thank individually but they can be assured of my profound gratitude.

Contents

Chapter 1

Introduction



"Nature does nothing uselessly."

- Aristotle

Humans have always been fascinated by nature. We feel a tremendous curiosity to understand nature and somehow try to copy what humans cannot do, but other species can. Humans have thus devised artificial artifacts to try to enhance their own capabilities: to let humans move faster, see farther, communicate over large distances, explore the ocean depths, and even fly. However, in some cases, copying nature has not been a very fruitful approach: when humans abandoned the idea of wings, they managed to fly faster than birds.

What we should do in certain cases is therefore to consider nature as a source of inspiration and, basically, try to understand and conceptualize the desired overall feature or behavior. Living organisms, for example, are complex systems exhibiting a range of characteristics very interesting to engineers, such as evolution, fault tolerance, and adaptation. Consequently, engineers have tried to develop bio-inspired computing machines by drawing their inspiration from three levels of organization, observed in nature: phylogeny, ontogeny, and epigenesis, featuring the capabilities of evolution, growth, and adaptation, respectively. In analogy to nature, a space of bio-inspired hardware systems has been defined along these three axes of organization, giving rise to the so-called *POE model* [193].

The focus of this thesis can be found along the epigenetic axis of the POE model. We aim to develop bio-inspired digital systems endowed with the capability of adaptation (learning) by exploiting the overall principles of operation of the brain and the nervous systems. In particular, we have been motivated by the idea of developing digital artificial neural networks with online learning capabilities. We argue that true learning systems adapt to problems by changing the strength of the interconnections between their computational elements (as is the case in most neural network models) and by dynamically reconfiguring their structure.

In this introductory chapter, we will present a short description of the POE model to place the present thesis within the domain of bio-inspired hardware systems (Section 1.1). We will then introduce the main goals of the thesis (Section 1.2), to conclude the chapter with a brief outline of the overall structure of the thesis (Section 1.3).

1.1 The POE model

The POE model of bio-inspired hardware systems assumes the existence of three axes of organization of life: phylogeny, ontogeny, and epigenesis [193].

- **Phylogeny:** The first axis concerns the temporal evolution of the genetic program, the hallmark of which is the evolution of species, or phylogeny [48]. The multiplication of living organisms is based upon the reproduction of the genetic program (DNA), subject to an extremely low error rate at the individual level, so as to ensure that the identity of the offspring remains practically unchanged. Mutation (asexual reproduction) or mutation along with recombination (sexual reproduction) give rise to the emergence of new organisms. The phylogenetic mechanisms are fundamentally nondeterministic, with the mutation and recombination rate providing a major source of diversity. This diversity is indispensable for the survival of living species, for their continuous adaptation to a changing environment, and for the appearance of new species.

  Bio-inspired phylogenetic systems include the so-called *evolvable systems*: systems that, starting with a set of potential solutions to a problem, mimic the process of reproduction to generate potentially better solutions and that of natural selection to keep the better solutions and discard the bad ones. This process has served as the inspiration for engineering techniques like *genetic algorithms* [87], *genetic programming* [110], etc. These ideas have been the inspiration for the development of an evolving hardware system (the *Firefly* machine) implementing online autonomous evolution [68].

- **Ontogeny:** Upon the appearance of multicellular organisms, a second level of biological organization manifested itself. The successive divisions of the mother cell, the zygote, with each newly formed cell possessing a copy of the original genome, is followed by a specialization of the daughter cells depending on their surroundings, i.e., their position within the ensemble as well as the milieu. This latter phase is known as cellular differentiation. Ontogeny is thus the developmental process of a multicellular organism. This process is essentially deterministic: an error in a single base within the genome can provoke an ontogenetic sequence which results in noticeable, possibly lethal, malformations.

  The main process involved in the ontogenetic axis can be summed up as growth or construction. Ontogenetic hardware exhibits properties such as replication and re-generation which find their use in many applications. For example, replicating systems have the ability to self-repair upon suffering heavy damage [203]. Replication can in fact be considered a special case of growth, involving the creation

of an identical organism by duplicating the genetic material of a mother entity onto a daughter one, thereby creating an exact clone. These ideas have been the inspiration for the development of *embryonic* hardware with self-repair capabilities [129, 131, 203].

- **Epigenesis:** The ontogenetic program is limited in the amount of information that it can store, implying that the complete specification of the organism is impossible. A well-known example is that of the human brain, consisting of approximately 10^{10} neurons and 10^{14} connections, far too large a number to be completely specified in a genome of approximately 3×10^9 characters in a 4-letter alphabet. Therefore, upon reaching a certain level of complexity, there must emerge a different process to allow the individual to integrate the vast quantity of interactions with the outside world. This process is known as epigenesis and primarily includes the nervous system, the immune system, and the endocrine system. These systems are characterized by a basic structure that is entirely defined by the genome (the innate part), which is then subjected to modification through lifetime interactions of the individual with the environment (the acquired part). The epigenetic processes can be loosely grouped under the heading of *learning systems*. Engineers have centered their attention on the nervous system, giving rise to the field of artificial neural networks, which will be the focus of this thesis.

1.2 Aims of the thesis

From a biological point of view, there is a growing evidence that the genome contains the formation rules that specify the outline of the nervous system [191]. It is primarily in the synapses, the zones of contact between two neurons, where learning occurs through interactions with the environment during the organism's lifetime. The nervous system of living organisms thus represents a mixture of the innate and the acquired, the latter precluding the possibility of its direct hereditary transmission under Darwinian evolution. A predominant approach in the field of artificial neural networks consists of using a database to train the system by applying a so-called *learning algorithm* to modify the interconnection dynamics between artificial neurons, using a predetermined network topology (i.e., the number of artificial neurons and the way they are interconnected). When the training process ends, the system remains fixed, and can be considered a *learned system*.

However, in applications where the entire training database is not available or may change in time, the learning process must persist, giving rise to *learning systems*. A problem of such systems is how to preserve what has been previously learned while continuing to incorporate new knowledge. Learning systems overcome such problems by a local adaptation process and by offering the possibility of dynamically modifying the network's topology.

Learned and learning systems

One can view a predesigned network as an implementation of a learned system that exhibits instinctive behavior [193]. Indeed, it seems that the human brain has many more such instinctive networks than is usually acknowledged [43, 215], possibly due to their being faster and less resource-intensive than *learning systems*, which adapt continuously within a dynamic environment. Learning systems exhibit the plasticity necessary to confront complex, dynamical tasks and must be able to adapt at two distinct levels, changing the dynamics of interneuron interactions as well as modifying the network topology itself. Topology modification has proven to be a successful solution to a problem known as the stability-plasticity dilemma: how can a learning system preserve what it has previously learned, while continuing to incorporate new knowledge [37]. We are particularly motivated by the idea of developing learning systems instead of merely learned ones.

Structure-adaptable digital neural networks

Many artificial neural network implementations exist, mostly in software rather than in hardware, though only the latter concern us here. While neural network hardware appeared as far back as the 1980's [134], only recently have we seen the use of *reconfigurable hardware* (e.g., field programmable devices) for implementing artificial neural systems, initially taking advantage of their potential for rapid prototyping [20, 47] and then of their reconfigurability for density enhancement [55] and acceleration [20]. However, the reconfigurability of such devices has not been exploited to implement neural networks with in-circuit structure adaptation, the goal of our research.

Adaptive categorization and learning by interaction

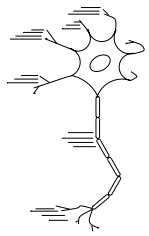
Categorization refers to the process by which distinct entities are treated as equivalent. It is considered as one of the most fundamental cognitive activities because categorization allows us to understand and make predictions about objects and events in our world. *Adaptive categorization* models have allowed us to explain some sensory and cognitive processes in the brain such as perception, recognition, attention, working memory, etc. However, the learning process of spatial and motor skill acquisition seems to be governed by a *learning by interaction* process. Adaptive categorization and learning by interaction are key features of an autonomous learning system: adaptive categorization is essential to make sense of the flood of environmental signals that relentlessly assails the sensors of a learning system, while learning by interaction is essential when an explicit external guide for adaptation is not available (i.e., a teacher) and the system has to learn by itself. We aim to develop a structure-adaptable digital neural network exhibiting these two key features. The resulting neural-based system should allow the control of an autonomous robot interacting with a real workspace. In particular, a real robot navigation task will allow us to validate the implementation of an on-chip learning system.

1.3 Outline of the thesis

This thesis is organized in 7 chapters, the first Chapter being this introduction. In Chapter 2 we introduce artificial neural networks, massively parallel computing machines inspired by biological nervous systems, focusing particularly on the problem of topology adaptation, which will be important for the subsequent discussion. Chapter 3 presents the key topics of neural hardware. We concentrate on digital hardware, and more particularly on reconfigurable hardware, a technology that allows users to reconfigure the internal circuit connections and node logic functionality. We argue that such technology should enable the implementation of true learning systems. In Chapter 4 we present a structure-adaptable artificial neural network architecture called FAST for *flexible adaptable-size topology*. Such neural architecture is an adaptation of the principles of the adaptive resonance theory (ART) [70, 71], a theory of human cognitive information processing. FAST was particularly motivated by the possibility of a digital implementation using programmable hardware devices, to be used for image processing and neurocontrol. Chapter 5 presents the problem of learning by interaction. The basic approach is to learn to make predictions about the consequences of behavioral responses in situations that resemble those experimented in the past. In particular we will describe a computational approach called temporal-difference learning, which belongs to a wider class of adaptive techniques called reinforcement learning algorithms. We explore the use of such techniques for learning strategies in gaming and control problems, compare their advantages and disadvantages, and propose a digital implementation of a neurocontroller. Chapter 6 presents the problem of controlling an autonomous mobile robot interacting with a real workspace. The robot navigation task will allow us to validate a neurocontroller architecture for on-chip learning systems. We also present the development of an FPGA-based board adapted for the control of the Khepera mobile robot, implementing the neurocontroller architecture. Finally, Chapter 7 presents some conclusions and discusses the goals achieved with the present work. It states the original contributions, and describes the limitations of the present system and proposes possible extensions and future work.

Chapter 2

Artificial Neural Networks



"If I have seen further than other men, it is because
I have stood upon the shoulders of giants."

-I. Newton

Recent advances in technology have enabled us to use modern digital computers capable of astounding feats. They operate on the nanosecond time scale and perform enormous and complex arithmetic calculations. Furthermore, they are able to store huge amounts of data: documents, images, scientific data, etc. Microprocessor designs will soon surpass 10 million transistors [211], a complexity hardly imagined in the middle 1950s when John von Neumann described the revolutionary architecture of modern digital computers as *an extremely simplified model of the living brain* [221].

Von Neumann's work on the *stored program* computer architecture (nowadays known as the von Neumann architecture) was based on the original ideas of Alan Turing who also wanted to *build a brain*. Turing thought the brain worked by changing *states*, thus he imagined a step by step programming machine [214].

The pioneers of informatics were clearly inspired from neurology and current knowledge of the operation of the human brain to develop the architectures of modern computing machines, even though the long course of evolution has given the brain very distinctive characteristics like learning, distributed memory and computation, generalization, robustness, the capability of interpretation of imprecise and noisy information, etc., not present in von Neumann computers.

Neuroengineers have came up with the idea of *artificial neural networks*, massively parallel computing machines also inspired by biological nervous systems, but based on a very different approach and a new paradigm, where *learning from examples* or *learning from interaction* replaces programming.

2.1 Biological neural networks

About 100 years ago, a Spanish histologist, Santiago Ramon y Cajal, the father of modern brain science, realized that the brain was made up of discrete units he called *neurons*, the Greek word for nerves. He described neurons as polarized cells that receive signals via highly branched extensions, called *dendrites*, and send information along unbranched extensions, called *axons* [60] (Figure 2.1).

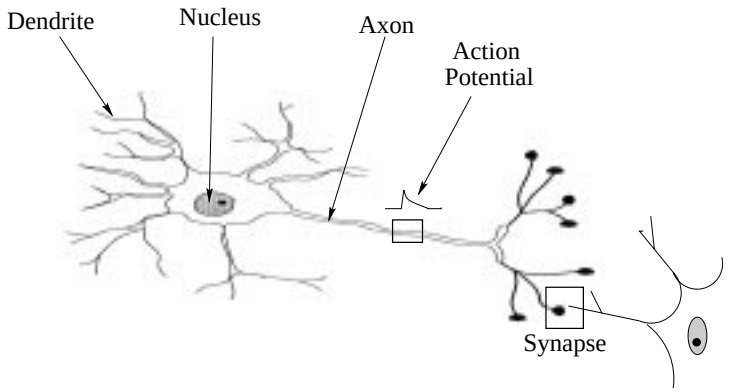


Figure 2.1: A biological neuron.

The brain is a very complex machine. The human brain contains between 7×10^{10} and 8×10^{10} neurons [185], which is on the same order of magnitude as the number of stars in our galaxy, and each neuron is connected up to 10^3 to 10^4 other neurons. In total, the human brain contains approximately 10^{14} to 10^{15} interconnections [100].

Although the brain exhibits a great diversity of neuron shapes, dendritic trees, axon lengths, etc., all neurons seem to process information in much the same way. Information is transmitted in the form of electrical impulses called *action potentials* via the axons from other neuron cells. These action potentials have an amplitude of about 100 millivolts, and a frequency of approximately 1 KHz. When the action potential arrives at the axon terminal, the neuron releases chemical *neurotransmitters* (from the synaptic vesicle) which mediate the inter-neuron communication at specialized connections called *synapses* (Figure 2.2).

These neurotransmitters bind to receptors in the membrane of the post synaptic neuron to excite or inhibit it. A neuron may have thousands of synapses connecting it with thousands of other neurons. The resulting effect of all excitations and inhibitions reduces the external membrane's electrical potential difference by about 70 millivolts (the inner surface becomes negative relative to the outer surface). Therefore, the permeability to sodium (Na^+) increases, leading to the movement of positively charged sodium from the extracellular fluid into the cell's interior, the *cytoplasm*, which in turn may generate an action potential in the post synaptic neuron (i.e., the neuron fires).

This very brief description of the neuron's physiology has inspired engineers and scientist to develop adaptive systems with learning capabilities. In the following section we will describe the main computational models that have been developed so far, as a result of such biological inspiration.

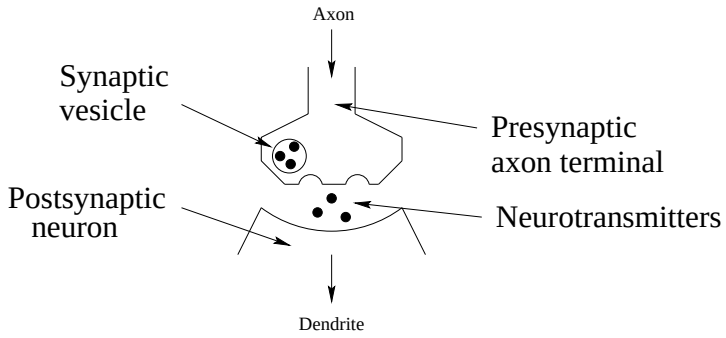


Figure 2.2: Synapses are specialized connections between neurons where chemical neurotransmitters mediate the inter-neuron communication.

2.2 Artificial neural networks

Artificial neural network models are widely used for the design of adaptive, "intelligent" systems since they offer an attractive paradigm: "learning" to solve problems from examples and from interaction. These models achieve good performance via massively parallel networks composed of generally non-linear simple computational elements, often referred to as *units* or *neurons*. An output value, referred to as the neuron's *activation*, is associated with each neuron. Similarly, a value, known as the *synaptic weight* is associated with each connection between neurons. The activation and the synaptic weight are analogous, respectively, to the firing rate of a biological neuron and the strength of a synapse (connection between two neurons) in the brain. In a network, the neuron's activation depends on the activations of the neurons connected to it and the interconnection weights. Neurons are often arranged as layers, where the activations of the neurons of the input layer are set externally (Figure 2.5).

Artificial neural networks are defined by the features of their neurons (i.e., the artificial neuron model), by their network topology, and by their training or learning algorithms. In the following sections we describe the main models of artificial neurons, the standard artificial neural network topologies, and the principal classes of learning algorithms.

2.3 Artificial neurons

The first computational model of the neuron was introduced with the pioneering work of Warren McCulloch and Walter Pitts [133] in the 1940s. McCulloch and Pitts merged mathematical logic and neurophysiology to propose a binary threshold unit as a computational model for an artificial neuron operating in discrete time. The output $y(t)$ of a neuron is 1 when an action potential is generated, and 0 otherwise. A *weight* value w_i is associated to each ith connection to the neuron. Such weights characterize the connections (synapses) as *excitatory* if $w_i = +1$, and *inhibitory* if $w_i = -1$. A neuron "fires" when the effect of inhibitions and excitations is larger than a certain *threshold* θ.

In 1958, the American psychologist Frank Rosenblatt proposed a computational

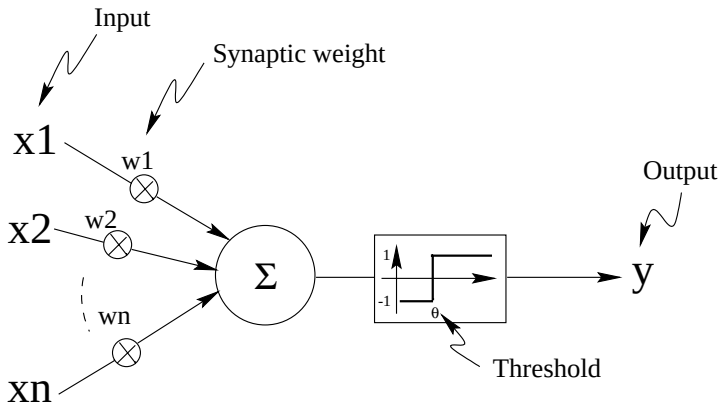


Figure 2.3: Model of an artificial neuron.

model of neurons he called the *Perceptron* [171]. The essential innovation was the introduction of numerical interconnection weights instead of the simple inhibitory/excitatory connections of the McCulloch-Pitts model (Figure 2.3). The model is mathematically described as follows:

$$y(x) = \begin{cases} +1 & if \ \sum_{i=1}^{n} w_i x_i > \theta \ , \\ -1 & if \ \sum_{i=1}^{n} w_i x_i < \theta \ , \end{cases} \tag{2.1}$$

where y is the output of the Perceptron, w_i is the weight of input x_i, and θ is the threshold. The inputs $(x_1, x_2..., x_n)$ and weights $(w_1, w_2..., w_n)$ are, typically, real values. Such weights model *synaptic efficacity*, and the output of such a neuron model the *rate of fire* of biological neurons. One can distinguish three types of artificial neuron models: input-correlation neurons, weightless neurons, and spiking neurons.

2.3.1 Input-correlation neurons

From a computational point of view, an artificial neuron either classifies or computes some function of the inputs. To determine its output for a given input, a neuron computes some sort of correlation between its inputs and the set of stored interconnection weights (i.e., the synaptic weights) [241]. Typically, this correlation is computed as a scalar product, which is a trivial measure of similarity [107]: if x_i and w_i are the ith components of the input and the weight vectors respectively, then the scalar product is equivalent to the standard sum of weighted inputs:

$$\sum_i w_i x_i \tag{2.2}$$

In an artificial neuron, this correlation value is referred to as the *potential* of the neuron. The output of the neuron is a non-linear function of the potential:

$$y = f(\sum_i w_i x_i) \ , \tag{2.3}$$

where f is typically a sigmoidal function or the hyperbolic tangent:

$$f(x) = (1 + e^{x})^{-1} or f(x) = (1 - e^{-x})(1 + e^{-x})^{-1} \tag{2.4}$$

A second method of forming statistical correlations is by means of *distance* computations. *Distance* and *similarity* are reciprocal concepts: we may call distance *dissimilarity* [107]. The best known measure of distance is the *Euclidean* distance, given by:

$$\sqrt{\sum_{i}(x_i - w_i)^2} \tag{2.5}$$

The *Minkowski* metric is a generalization of the Euclidean distance:

$$(\sum_{i}(x_i - w_i)^n)^{1/n} \tag{2.6}$$

The *Manhattan* or *city-block* metric is the Minkowski distance with $n = 1$:

$$\sum_{i} \|x_i - w_i\| \tag{2.7}$$

A complete review of similarity metrics discussing not only real-valued magnitudes but discrete-values, scalar values, and orientation instead of magnitudes can be found in [107].

2.3.2 Weightless neurons

A *weightless* or a *logical* neuron is defined as an elemental computing device which learns a combinatorial mapping between an input binary n-tuple and a single binary output [5]. Weightless neurons are also known as RAM-based neurons [17], since they are implemented using random-access memories (RAM). These models use the *Hamming* distance (i.e., the number of bits by which a pair of bit-strings differ) as the measure of similarity to correlate the inputs to the neuron and a *stored* value. The generalizing random-access memory or G-RAM neuron model [5] is an example of such neuron: given a set of training pairs $T = \{(\mathbf{I}_1, z_1), (\mathbf{I}_2, z_2)...(\mathbf{I}_n, z_n)\}$ of an unknown input vector $\mathbf{I}_u$, the response is obtained by comparing the input vector with those in the training set using the Hamming distance. The corresponding output to $\mathbf{I}_u$ is then the output of the neuron for the input vector $\mathbf{I}_i$ that has the minimum Hamming distance with $\mathbf{I}_u$. If several vectors in the training set satisfy this condition but differ in the corresponding outputs, the output of the neuron is chosen to be 0 or 1 with, for example, a probability of 0.5 (Figure 2.4).

Variants of weightless neurons output probabilistic values to achieve a certain type of generalization called *nearest-neighbor* retrieval in patter recognition applications [5].

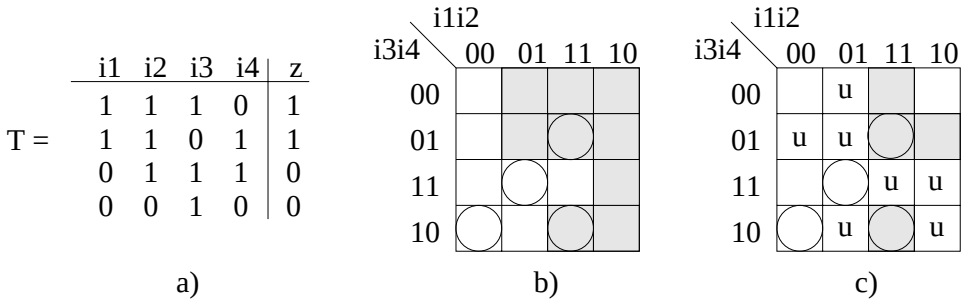


Figure 2.4: Example of generalization of a G-RAM weightless artificial neuron. a) The training set T. b) Karnaugh-map representation of the generalization in a weighted system: the circles are the training patterns, and the shaded squares represent the generalized outputs (instances where $z = 1$). c) Generalization in a weightless G-RAM system: **u** represents an 'undefined' output; the corresponding value is a 1 with a probability 0.5.

2.3.3 Spiking neurons

It has been found that biological neurons in the cortex fire at various intermediate frequencies between a maximum and a minimum frequency. This is why the output of a sigmoidal Perceptron is interpreted as the current firing rate of a biological neuron. However, experimental evidence has shown that biological neural systems use the timing of a single action potential (a spike) to encode information. Therefore, a mathematical model for *integrate and fire* neurons or *spiking neurons* has been developed [65]. A network of such spiking neurons has proven to be at least as powerful as a sigmoidal Perceptron with simpler circuits. Spiking neurons are considered the third generation of neural models after the McCulloch-Pitts neurons and the neurons with weighted sum of inputs and continuous activation functions [125].

2.4 Artificial neural network topologies

The simplest form of an artificial neural network is the single-layer *Perceptron* (Figure 2.5a). The idea of cascading layers of Perceptrons may come originally from the knowledge of the organization of the brain, where layering is a common architectural feature [128]. However, it was not until the rediscovery of the *backpropagation* algorithm (first published in 1974 in J.P. Werbos PhD thesis *Beyond Regression* [224]) by Rumelhart, Hinton, and Williams [174] that Multi-Layer Perceptrons (MLP) and, in general, multi-layer neural networks (Figure 2.5b) found their place in the history of connectionist systems.

Another kind of topology for neural networks was introduced by Hopfield in the early 1980s as a model of memory [88]. These neural architectures possess recurrent connectivity (Figure 2.5c), which provides a sort of short-term memory and a dynamic behavior. Eventually, they find a stable state that allows them to store information. Such networks can be used as *associative memories* which are *content addressable*, i.e., a memorized pattern can be retrieved by specifying a part of it.

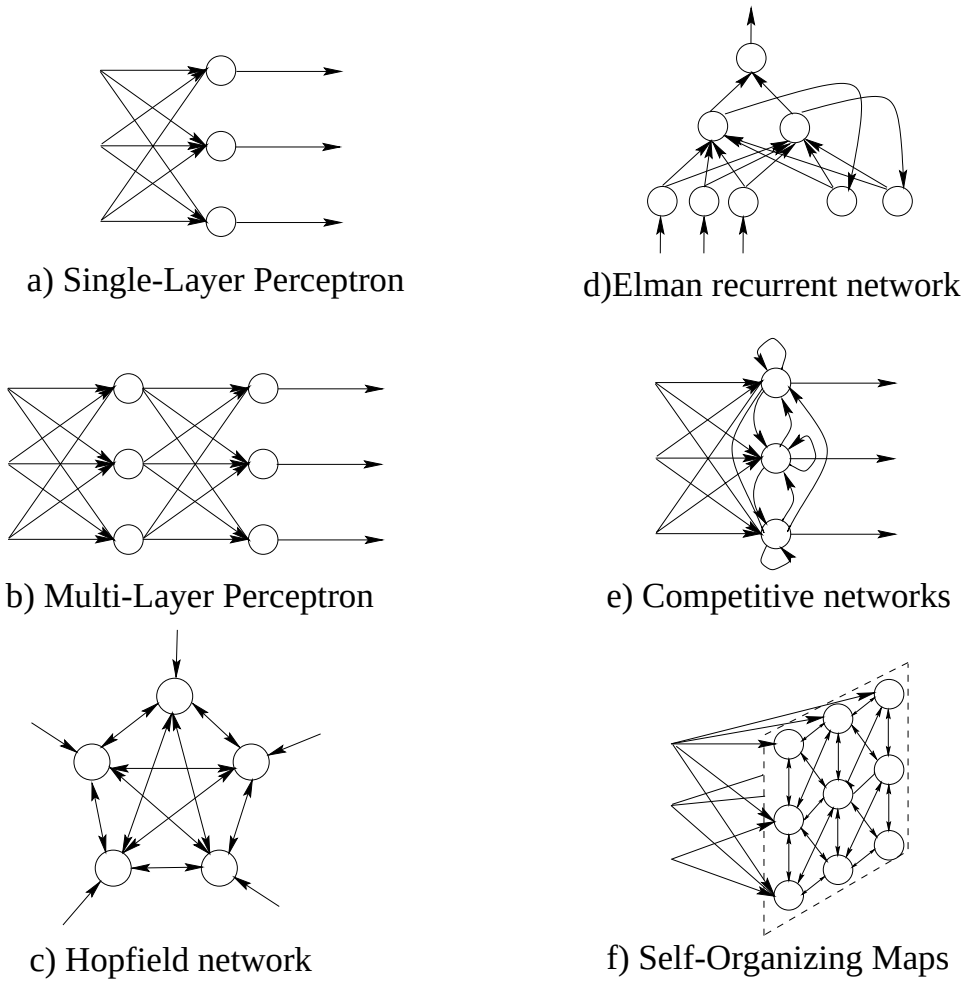


Figure 2.5: Typical artificial neural network topologies.

Similarly, artificial neural networks with recurrent connections behaving somewhat like sequential logic circuits have also been developed (Figure 2.5d for a particular example). These networks have a sort of short-term memory and previous experiences may have an effect on the activation of such units. They are widely used when systems display temporal variation and temporal context dependence [108].

Finally, the so called *competitive* neural networks and *self-organizing maps* (Figures 2.5e and 2.5f respectively) exhibit a special arrangement of self-excitatory and lateral inhibitory connections that enable them to *self-organize* and build input feature maps [107].

2.5 Learning in artificial neural networks

Learning is often associated with increase of knowledge or understanding, and it can be achieved by self-teaching, instruction, or experience. Nevertheless, it is very difficult to precisely define learning, and even more difficult to determine if a machine is capable of

learning.

The English mathematician Alan Turing was perhaps the first person to propose the challenging question "Can machines think?" in his seminal article *Computing machinery and intelligence* [214]. Even though he predicted, in the same article, that "at the end of the century one will be able to speak of machines thinking without expecting to be contradicted", this subject is still very controversial.

Undoubtedly, this controversy and the surprising advances in our understanding of the brain turn this area of research in a really fascinating one. Providing not only new engineering solutions but also new ways to approach philosophical questions about ill-defined concepts like autonomy, learning, intelligence, intentionality, consciousness, etc.

Norbert Wiener, the father of *cybernetics*, presented a very general but structured definition of learning:

> *"To begin with learning machines: an organized system may be said to be one which transforms a certain incoming message into an outgoing message, according to some principle of transformation. If this principle of transformation is subject to a certain criterion of merit of performance, and if the method of transformation is adjusted so as to tend to improve the performance of the system according to this criterion, the system is said to* learn*"* [231].

In the artificial neural network context, learning is thus closely related to an improvement of performance of the system. This is achieved by minimizing an error metrics, by self-organizing information through correlations of the data, or by maximizing rewards in a trial-and-error system over time. In practice, learning in artificial neural networks is achieved by a proper adjustment of the interconnection weights between artificial neurons and sometimes by topology adaptation [153].

A *learning algorithm* refers to a procedure in which learning rules are used for adjusting the weights of an artificial neural network, and possibly its topology, i.e., the number of layers, the number of neurons, and the pattern of connections. So far, the only biologically realistic learning rule is the well-known *Hebb's rule*, which states that "when two interconnected neurons fire at the same time, and repeatedly, the synapse's strength is increased" [83].

Hebb's rule has inspired a large number of learning algorithms which can be classified into three main learning paradigms: supervised, unsupervised, and reinforcement learning.

2.5.1 Supervised learning

In *supervised learning* or *learning with a "teacher"*, the training inputs are provided together with the desired outputs. A basic principle of this kind of learning algorithms is *error correction*: an error value is generated from the actual response of the network and the desired response, then, the weights are modified so that the error is gradually reduced. To solve the supervised learning problem, two steps are required: first, we must specify the *topology* of the network and, second, we must specify the *learning rule*.

Currently, *backpropagation* [224] is the most popular learning algorithm for artificial neural networks. It is used in about 70% of real-world applications [225]. Nevertheless, a vast amount of research aims at improving its generalization (i.e., the ability to predict an output for a previously unseen input), its learning speed, and its fault-tolerance (i.e., the ability to perform well in spite of noise or faults).

Supervised neural networks have been used in a wide range of applications which basically fall into three categories: (1) classification and diagnosis, (2) function approximation, compression, feature extraction, and quantization, and, (3) optimization. This approach can be generally seen as a *learned* one since the network does not change after its training [193]. Our work is mostly related to the other two learning paradigms: reinforcement and unsupervised learning.

2.5.2 Reinforcement learning

Reinforcement learning is a synonym of *learning by interaction*: during learning, the adaptive system *tries* some actions (i.e., output values) on its environment and it is *reinforced* by receiving a scalar evaluation (the reward) of these actions. The reinforcement learning algorithms selectively retain the outputs that maximize the received reward over time. Reinforcement learning clearly differs from supervised learning methods where explicit input-output examples are presented each time to the system during learning. The two basic concepts behind reinforcement learning are *trial and error search* and *delayed reward*. See Chapter 5 for a more detailed description of this methods.

Reinforcement learning techniques are straightforward approaches to implementing real online systems capable of learning from experience. It has been widely used in learning games (Backgammon [204], Chess [114], Checkers [176], etc), control problems, dynamic channel allocation in cellular telephone systems, packet routing in dynamically changing networks, improving elevator performance [200], learning autonomous agents and robots (Chapter 6), etc.

2.5.3 Unsupervised learning

Unsupervised learning neural networks *cluster*, *code*, or *categorize* input data: similar inputs are classified as being in the same category, and should activate the same output unit, corresponding to a prototype of the category. Clusters are determined by the network itself, based on correlations of the inputs [85].

A basic principle of unsupervised learning is competition: output units compete among themselves for activation. As a result, in most competitive learning algorithms only one output neuron is activated at any given time. This is achieved by means of a so called *winner-take-all* operation, which has been found to be biologically plausible [85].

These techniques allow the implementation of very powerful feature extraction modules for autonomous learning systems, as we will show in Chapter 4. Moreover, they have been widely used in clustering tasks, data dimensionality reduction, data mining (data organization for exploration and search), information extraction, density approximation, data compression, etc. [85].

2.6 Artificial neural network topology adaptation

As we have seen in the last section, most neural network models base their ability to adapt to problems on changing the strength of their interconnections according to the learning algorithm. However, the lack of knowledge in determining the appropriate topology, including the number of layers, the number of neurons per layer, and the interconnection scheme, often sets an *a-priori* limit on performance. For example, if a network is too small, it does not accurately approximate the desired input to output mapping. This problem is known as *oversmoothing*. If a network is too large, it requires longer training times and may not *generalize* well, that is, it may correctly map the training examples but give incorrect outputs at other points (previously unseen, but related to the same problem). This problem is known as *overfitting*. Figure 2.6 illustrates this phenomenon: as time passes, the training error diminishes, whereas the test error increases [166].

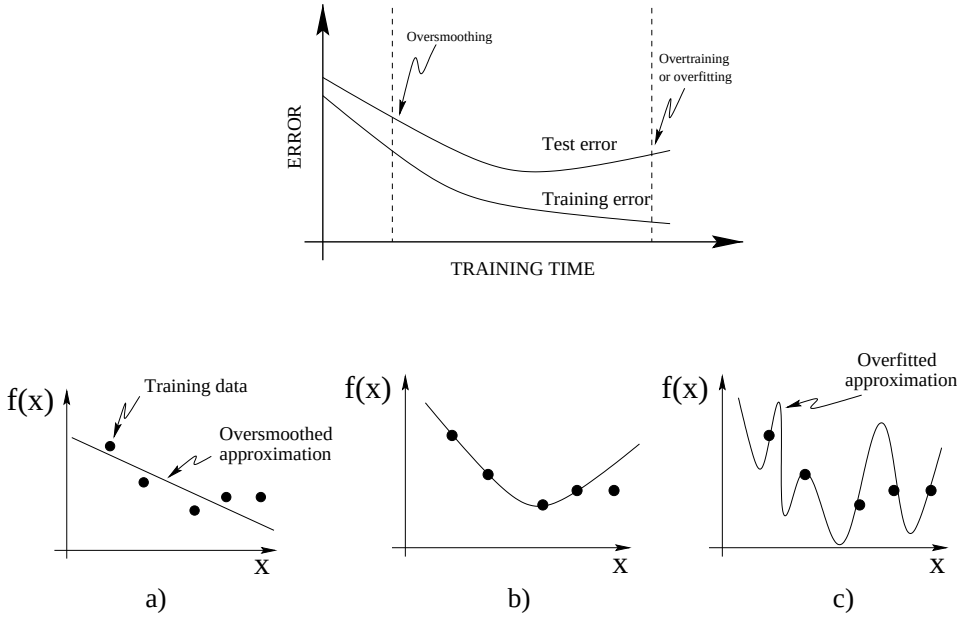


Figure 2.6: Evolution of the error of the system over the training and the test patterns in time. In a) we present the problem of oversmoothing (poor approximation), b) presents a "good" approximation, and c) the problem of overfitting (loss of generalization ability).

It is not obvious how to determine a good neural network topology for a given task, i.e., how to successfully manage the *bias-variance dilemma* [180] (the tradeoff between overfitting and oversmoothing), and the *stability-plasticity dilemma* [37] (how can a learning system preserve what it has previously learned, while continuing to incorporate new knowledge). Theoretical works trying to determine the structure of an artificial neural network include the use of the so called *Vapnik-Chervonenkis* dimension (or simply VC-dimension), which is defined as the maximum size of a set of examples *shattered* by the neural network (a set of examples S is said to be shattered by a neural network if it realizes all the dichotomies of S), and which in many cases can be related to the structure of the network [124]. Other approaches include *modular neural networks* that

attempt to overcome these problems by allowing different types of neural networks to be connected as modules of a whole network. Finally, two different strategies have been used to determine or adapt the topology of artificial neural networks: the use of evolutionary techniques and the development of *ontogenic neural architectures*.

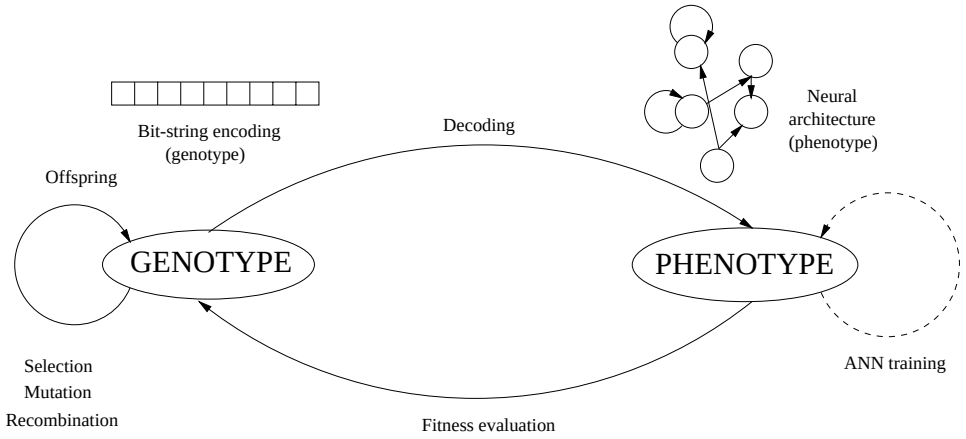


Figure 2.7: Evolutionary design of artificial neural networks. A set (population) of bit-strings (genomes) used to encode possible artificial neural network topologies is first generated at random. At every evolutionary step (generation) the different genomes are decoded to construct artificial neural networks to be tested on a given problem. An evaluation (fitness) is obtained from such test(s). To form a new generation of possible neural topologies (solutions), individuals (genomes) are selected according to their fitness and modified by genetic operations (mutation and crossover) until a good solution is found.

2.6.1 Evolutionary design of artificial neural networks

Evolutionary techniques such as genetic algorithms have also been proposed for the design of artificial neural networks [21, 148, 240]. Both the network weights and the topology can be coded into bit-strings, which are then "evolved" with such algorithms (Figure 2.7). However, since one of our objectives is digital implementation and we are particularly interested in achieving online adaptation, it would be very difficult to implement a population of neural networks: at best, each individual neural network would be implemented sequentially, rendering the topology adaptation an off-line task [155]. Furthermore, this kind of approach requires that every neural architecture (the phenotype) be tested in order to determine the *fitness* of the corresponding bit-string encoding (the genotype), a computationally expensive task.

To overcome the problem of implementing populations of artificial neural networks some researchers have developed learning algorithms that not only adapt the interconnection weights between neurons, but also activate new neurons and create or delete interconnections between neurons within an individual artificial neural network. Some of those algorithms were improperly called *evolvable artificial neural networks* [84, 146, 155].

A special class of learning algorithms was thus introduced to overcome the above problems, offering the possibility of dynamically modifying the network's topology, thus

providing *constructive incremental learning*. Artificial neural networks making use of these kind of learning algorithms have been called *ontogenic artificial neural networks* as an analogy between the structural development of the networks and the development of individual organisms (i.e., *ontogeny*).

2.6.2 Ontogenic artificial neural networks

Ontogenic artificial neural networks are networks that exploit incremental learning algorithms to modify both the interconnection weights between neurons and the topology of the network itself [59].

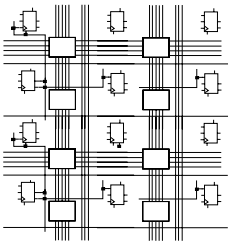
Weight adaptation is achieved using standard learning rules (e.g. Hebbian learning-like rules), while topology adaptation is achieved through two mechanisms: *growth* and *pruning*. The former typically starts with small networks and adds units or connections to the network to better approximate a given function. The latter starts with large networks and *prunes* or removes parts that are not needed, in order to shrink the networks and avoid overfitting. Ontogenic neural networks include supervised growing algorithms such as the *cascade-correlation* algorithm [57], the *Upstart* algorithm [63], and the *Meiosis* algorithm [16], unsupervised growing algorithms such as *ART* [37] and the *Growing Cell Structures* [64], pruning algorithms such as *weight decay*, *Optimal Brain Damage*, and *Optimal Brain Surgeon* [16].

2.7 Summary

Artificial neural networks are massively parallel computing machines inspired by biological nervous systems, where *learning from examples* and *learning by interaction* replaces programming in order to provide *adaptivity*. While learning is often associated with increase of knowledge or understanding, and can be achieved by self-teaching, instruction or experience, it is very difficult to precisely define. In the artificial neural network context it is closely related to proper adjustment of interconnection weights. We argue that learning is also closely related to topology adaptation, since the behavior of an artificial neural model is also very dependent on the network's topology, that is, the number of layers, the number of neurons, and the pattern of connections. A special class of learning algorithms has been introduced to overcome such problems by offering the possibility of dynamically modifying the network's topology, thus providing constructive incremental learning. Such networks have been called ontogenic neural networks, and are related to a wide variety of learning algorithms for both supervised and unsupervised learning. To our knowledge, there are no references to ontogenic reinforcement learning networks in the literature. However, we will see in Chapter 5 that some approaches imply an inherent structure adaptation in reinforcement learning networks.

Chapter 3

Neural Hardware



"Why cannot we write the entire 24 volumes of the Encyclopedia Britannica on the head of a pin ?"

-R.P. Feynman

Neuronal systems are difficult to simulate because they consist of a large number of generally nonlinear elements and have a wide range of time constants. The mathematical behavior of such systems can rarely be solved analytically [53]. However, typical artificial neural systems are very simplified versions of biological systems.

A neural network model can basically be implemented or simulated in three main ways: by programming a general-purpose computer, by programming a neural network dedicated machine, i.e., a *neurocomputer*, or by building a special-purpose device to implement a particular network and learning algorithm [13]. A general-purpose machine can simulate any neural network model and any learning algorithm, though the speed of such simulations tends to be slow. Dedicated neural network machines may work faster, but they are typically more difficult to program. Special-purpose hardware provides sufficient performance for real-time interaction, but it usually demands a significant investment [118].

It was often assumed, in the early years of neural network research, that implementation in special hardware would be required to fully exploit the networks' capabilities. Such hardware, in particular, would probably be analog and involve heavily interconnected multiple parallel *processing elements* (PE). However, the tremendous growth in the digital computing power of conventional von Neumann machines has allowed software neural network simulations to achieve great success in a number of applications [117]. Indeed, one of the problems facing the designer of neural network hardware is the Amdahl law, which states that parallelization is efficient only when a substantial part of a task can be parallelized. For example, speedups of 10 or more can be achieved only when 90% or more of the program execution can be made parallel [97, 117].

The first attempts to build special hardware for artificial neural networks date back to the 1950s. In 1951, Marvin Minsky built the first randomly wired neural network learning machine (called SNARC, for Stochastic Neural-Analog Reinforcement Computer), simulating adaptive weights with potentiometers. In the late 1950s, Rosenblatt implemented the first Perceptrons using variable resistors in much the same way as Minsky. In the early 1960s, Widrow and Hoff used custom elements called *memistors*, electrically variable resistors, to implement Adaline and Madaline networks [229]. In the late 1960s and 1970s, Aleksander proposed the first digital logic motivated learning networks modeling the neuron with a random access memory [4, 58]. These early weightless models evolved into the so called *RAM-based* neural networks [5, 17], and have ultimately served as a substrate to propose a theory of artificial consciousness [2, 3].

3.1 Neuromorphic and functional artificial neural networks

We can distinguish two classes of implementations of artificial neural networks: *neuromorphic* and *functional*. The former consists of those implementations that try to directly mimic the morphology and the physics of biological systems. The latter comprises implementations of bio-inspired engineering solutions, which try exploit the known advantages of more conventional engineering techniques.

Carver Mead at California Institute of Technology introduced the term *Neuromorphic Engineering* [134] for a new field based on the design and fabrication of artificial neural systems, such as vision systems, head-eye systems, and roving robots, whose architecture and design principles are based on those of biological nervous systems. Neuromorphic engineering has a wide range of applications from nonlinear adaptive control of complex systems to the design of smart sensors. Many of the fundamental principles in this field, such as the use of learning methods and the design of parallel hardware, are inspired by biological systems. These artificial neural models are composed of the so called *silicon neurons*, analog electronic circuits fabricated in complementary metal oxide semiconductor (CMOS) medium using very large scale integration (VLSI). CMOS is a technology commonly used to fabricate the digital circuits found in general-purpose digital computers. However, in silicon neurons, this technology is used to construct analog circuits whose physics is analogous of the physics of neural membrane conductivity [53].

Functional artificial neural networks need not to follow strictly the physics and organization of biological neural systems. Instead, they generally profit of the high-precision computation and the relatively small design effort of classical digital implementation techniques.

Analog vs. digital implementations

Analog implementations of artificial neural networks are extremely compact when compared with digital implementations, they are inherently parallel and their power consumption is very low. However, some controversy exists on the subject of precision and

weight storage in analog neural computation [53]. The analog alternative has been particularly successful on the design of neuromorphic systems such as silicon retinas and cochleas [134] and signal processing systems [219]. The most 'viable' hope for engineers seems to be the development of neuromorphic sensors that can feed their readings of the environment to digital electronics for subsequent processing [223] (Figure 3.1).

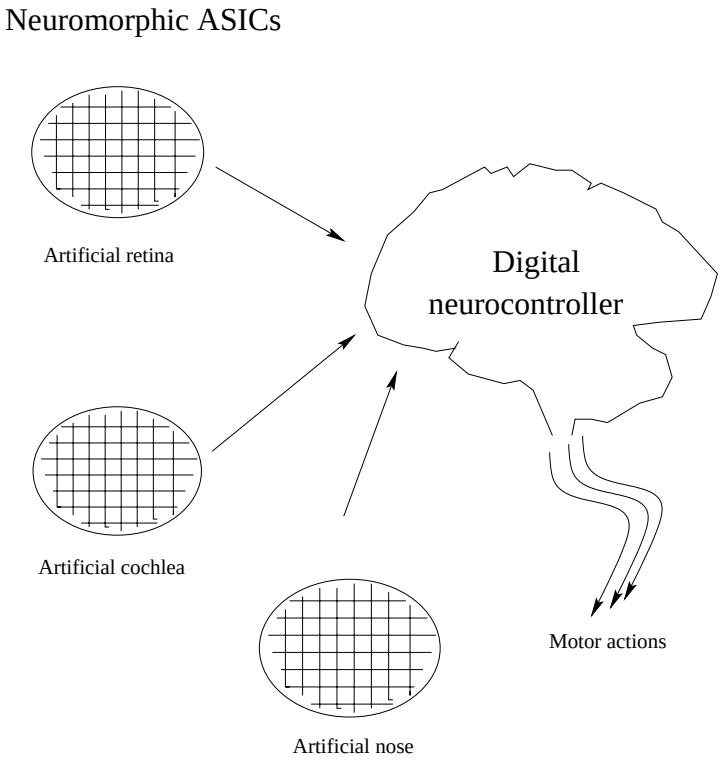


Figure 3.1: Mixed digital-analog artificial neural systems. Neuromorphic circuits sense the environment and feed their readings to digital electronics implementing *brain-like* computation.

3.2 Digital neural hardware

Digital implementation of artificial neural networks include the design of neural network dedicated machines (i.e., neurocomputers) and the design of special-purpose circuits to implement a particular network architecture and learning algorithm [13].

3.2.1 Digital neurocomputers

Digital neurocomputers are basically multiprocessor systems based on commercial or custom processors. In many cases, custom processors have been designed for artificial neural networks, even if they usually require a much more important development effort. Multiprocessor systems based on commercial processors have the advantage of shorter design time, lower cost, and software support. Furthermore, the constant advancement

of standard microprocessor chips and the emergence of new and faster algorithms tends to favor this last approach [98].

Two architectures dominate neuroprocessor designs: *single-instruction multiple data* (SIMD) and *systolic arrays* systems. The former executes the same instruction in parallel but on different data. The latter executes a step of a calculation before passing the results to the next processor in a pipelined manner. A complete review of the most common hardware architectures and neuroprocessor architectures can be found in the theses of Moreno [146] and Ienne [97].

3.2.2 Special-purpose digital neural hardware

Special-purpose digital neural hardware is used to solve problems with particular speed and/or area requirements. From the user's point of view we can distinguish three different approaches: *offline learning*, where learning (i.e., the modification of the weights of the learning system) occurs on a general-purpose machine before the learned system is implemented in the neural hardware, *chip-on-the-loop learning*, where the learning algorithm is executed on a general-purpose machine, but computations of the learning system (i.e., the sums of weighted inputs) is implemented using the neural chip, and *on-chip learning*, where the learning algorithm and the learning system are implemented on the neural hardware. In this work we are interested particularly in the on-chip learning approach.

3.3 Data representation for computation in artificial neural networks

There are two broad ways of representing values: analog and digital. In analog representation, the value is represented by continuous physical quantities. In the digital representation, the value is represented by a finite string of symbols (i.e., an alphabet) [178]. The most common alphabet used in computer science is composed of the symbols 0 and 1, the *binary* digital system. A 1−digit string in such alphabet is called a *bit*, and a string of several bits is normally used to represent numerical values. One can distinguish three kinds of data representation depending on the manner one interprets the string of bits that codes a numerical value: *positional*, *probabilistic*, and *redundant* representation.

3.3.1 Positional representation

In a positional representation, numeric values are represented as strings of digits. The sum of the values of all positions in the string equals the value of the string. In the binary number system, for example, each position has a weight that is two times greater than that of the position to the right, therefore, the binary number $b_n...b_2b_1b_o$ corresponds to

$$\sum_{i=0}^{n} b_i 2^i = b_0 2^0 + b_1 2^1 + b_2 2^2 + ...b_n 2^n \tag{3.1}$$

Three systems for representing numerical values in the binary positional encoding have been widely implemented in computer systems: *sign-and-magnitude*, *one's complement*, and *two's complement*. The most popular system for fixed-point number systems is the two's complement, while sign-and-magnitude numbers are more frequently used in floating-point systems.

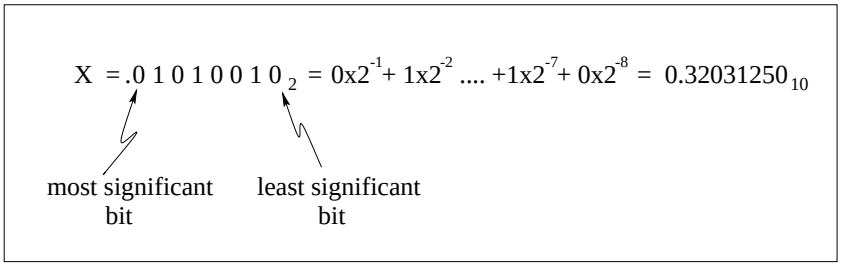


Figure 3.2: Example of positional value representation: 8-bit precision fixed-point representation of an unsigned real value $X \in [0, 1)$.

The possible operations on binary coded information are based on Boolean algebra, the mathematical framework defined by the English mathematician George Boole (1815-1864) [178]. Positional binary representations are the most widely used representations in digital computational systems, and thus in artificial neural network implementations.

3.3.2 Probabilistic representation

Inspired by biology and born out of a desire to perform analog computations using digital fabrication processes, the so-called *pulse stream* arithmetic systems have been steadily improved since their inception in 1986 [77]. One example of pulse stream arithmetic systems it the so called *stochastic logic* [242].

Stochastic logic is a digital technique which realizes pseudo-analog operations using stochastically coded pulse sequences. Computation is performed by manipulating streams of random bits which represent real values using a probabilistic coding: the number represented is the probability of a bit having the value '1'. If two real values are represented by two non-correlated random bit streams in this way, then the product of the two values will be implemented by the logical AND of the two bit streams.

This approach implies that the multiplication of real values can be achieved using a very small amount of logic. Furthermore, it has been found that the activation functions (i.e., a sigmoid) on every neuron can also be easily implemented using very simple digital circuits. Finally, the potential response speed of this technique is particularly interesting when considering the fact that successive layers in a network may be pipelined [192]. However, stochastic value representations require more memory than positional value representations: a n-bit stochastic bit-stream can represent only $n + 1$ values.

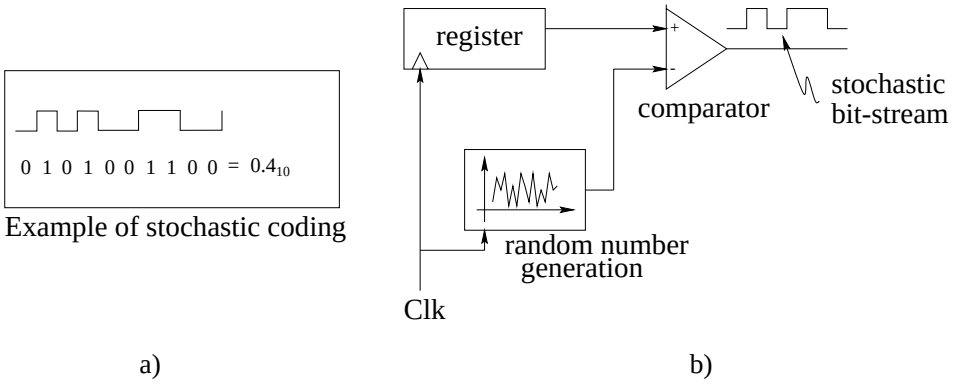


Figure 3.3: Example of stochastic value representation and generation of a stochastic bit-stream: a) a 10-bit stochastic stream represents a decimal value of 0.4 given that the probability of a bit having the value '1' is 40%. To generate such a bit-stream, the circuit in b) outputs a '1' with a probability that is a function of the value stored in the register.

In the artificial neural network domain, stochastic logic has been used mostly for learned systems, that is, systems where the training is implemented in software using a general-purpose machine, and the trained network in hardware [74]. However, it has been recently used to implement Hebbian learning based accelerators [62] and on-chip learning systems using standard digital VLSI technology because of its ability to escape from local minima [242].

Stochastic logic is a very promising technique. However, the requirement for independent random bit streams implies the need for many random sources. There have been several attempts to solve this problem by using techniques for generating multiple uncorrelated pseudorandom sources [7], but they require considerable hardware resources [28].

3.3.3 Redundant representation

Although probabilistic representations are inherently redundant (several bit-streams may represent the same decimal value), in this section we present another redundant numbering system which is deterministic. In contrast with the positional systems, in which the degree of redundancy is very small and seen as an inconvenience. Redundancy is the key feature of *redundant numbering systems*, allowing for rapid and compact implementations of arithmetic operators. In a non-redundant number representation with an integer radix r, a digit is allowed to assume r values: $0, 1, 2...r-1$. A number is represented in radix r as:

$$\sum_{k=-\infty}^{\infty} a_k r^{-k} \ , \tag{3.2}$$

where $a_k \in D_r = \{0, 1, ...r-1\}$. D_r is called the *digit set*.
In a redundant number representation, a digit is allowed to assume more than r values. Avizienis [18] represents redundant numbers in radix r using the digit set -a,-a+1,....a-1,a,

where $a \leq r - 1$.

In radix 2, $a_k \in \{-1, 0, 1\}$ (-1 is sometimes represented as $\bar{1}$).
To represent the digit set two bit values are needed:

$$a_k = (a_k^+ - a_k^-)$$

therefore:

$$\begin{aligned} 0 &= \quad (0,0) = (1,1) \\ 1 &= \quad (1,0) \\ \text{-}1 &= \quad \bar{1} = (0,1) \end{aligned}$$

and, in general, $a = \sum a_k 2^k = \sum (a_k^+ - a_k^-) 2^k$

Figure 3.4: Example of redundant value representation.

One of the main advantages of redundant representation is that sums can be performed without carry propagation [67]. Furthermore, redundant representations can be used with *on-line* arithmetics [56], a kind of serial computation where the most significant digits (MSD) are the first operands to be used (Figure 3.4), to achieve very compact implementations [67]. It is generally necessary to implement a decoding circuit to interface redundant logic operators with standard positional value representations systems, but, the overhead is not very significant when the number of arithmetic operations is high [67].

3.4 Architectures for dedicated neural hardware

Many possible ways to realize artificial neural networks using dedicated digital hardware have been proposed, and a number of studies have attempted to class such different approaches [97, 99, 118]. However, none of them is sufficient on its own [97]. A first classification of digital artificial neural network dedicated hardware is based on the number of processors being used to implement the accelerator or neurocomputer: *single processor* or *multiple processor* architectures [146].

According to Ienne [96], there are two main types of multiple processor architectures for artificial neural networks, depending on the particular data flow: *systolic* and *general parallel* architectures. While systolic architectures are typically designed to rhythmically compute very simple operations and pass data through the system [146], general parallel architectures are designed by dividing the system data flow into computational slots, which usually correspond to the time needed to process an input data vector [146].

Figure 3.5 presents some typical systolic array architectures classified according to the processing element interconnection network. Figure 3.6 shows a general parallel architecture with a broadcast bus. The architecture consists of a linear array of processing

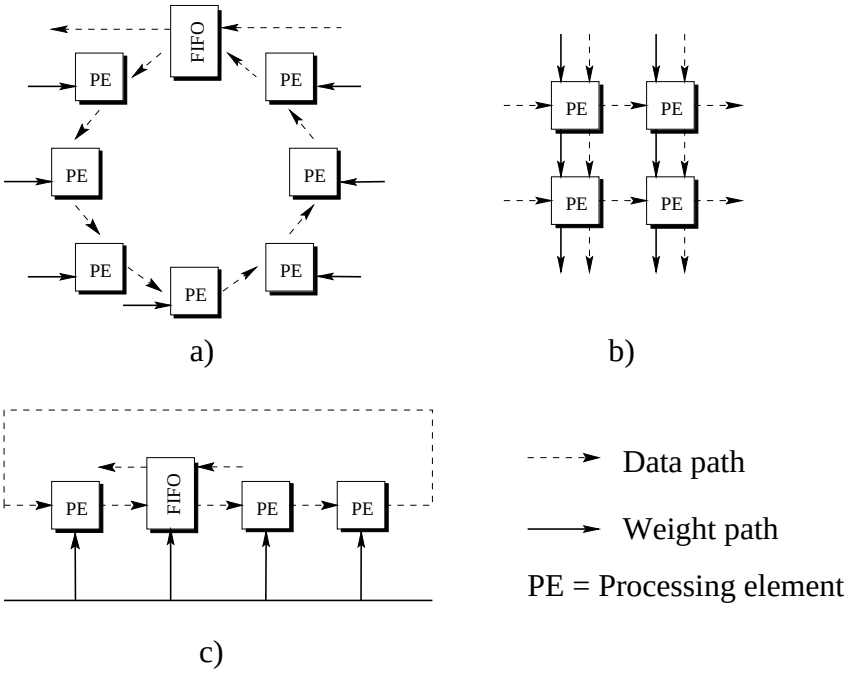


Figure 3.5: Typical systolic array architectures based on the processing element interconnection network: a) systolic ring, b) systolic square array and c) systolic ring with global bus architectures.

elements sharing a global input bus, and a global output bus. All these architectures usually present a one-to-one mapping of artificial neurons and processing elements.

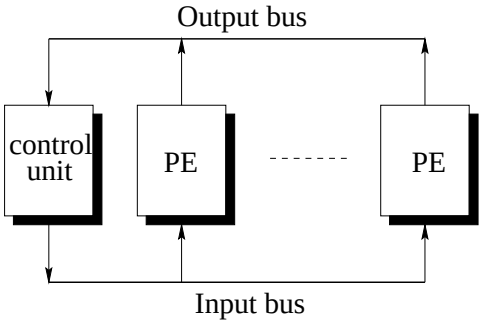


Figure 3.6: Broadcast bus general parallel architecture. (PE means processing element)

3.5 Reconfigurable neural hardware

Recent progress in semiconductor devices, and notably the development of configurable hardware devices such as *field-programmable gate arrays* (FPGA), allows users to reconfigure their internal circuit connections and node logic functionalities [177, 212]. An FPGA is an array of logic blocks placed within an infrastructure of interconnections,

and can be configured at three distinct levels: (1) the function of the logic block, (2) the interconnections between the blocks, and (3) the inputs and outputs. All three levels are configured via a string of bits that is loaded from an external source into configuration registers (Figure 3.7). This novel technology introduces a distinction between *programmable* processors and *configurable* devices. In the first case, a general-purpose processor is programmed by describing a given *algorithm* by means of a limited set of instructions. In the second case, an FPGA is configured by means of a bit-stream that completely specifies the logical functions and connectivity to be implemented, in order to realize a given design [179]. A configuration bit-stream needs not be written at very low level: as for compilation in the programmable paradigm, in configurable computing one can convert a high-level description of a design (specified, for example, using a high-level hardware description language like VHDL) can be converted into a configuration bit-stream by a process of synthesis, partition, placement, and routing (Figure 3.8).

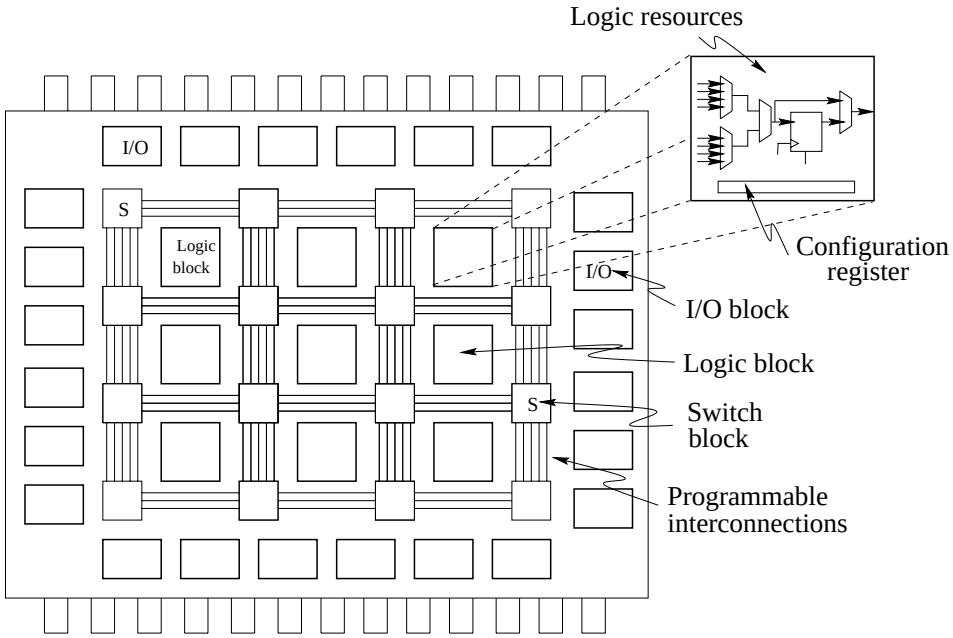


Figure 3.7: Typical field-programmable gate array (FPGA) architecture.

3.5.1 Field-programmable gate arrays (FPGA)

FPGA devices allow a very direct approach in which the hardware can be structured to directly implement the native operations required by the application and be organized to exploit the concurrency inherent in the computation. Such digital devices afford rapid prototyping and relatively inexpensive implementations. FPGAs have been typically used for prototyping ASIC (Application Specific Integrated Circuits) designs and as low-cost parts in applications where the expense of designing and fabricating an ASIC is not justified [217, 218]. While FPGAs are considerably slower than ASICs and need higher power requirements, they can achieve a speed-up of an order of magnitude in

applications characterized by deeply pipelined, highly parallel, and integer arithmetic processing [217].

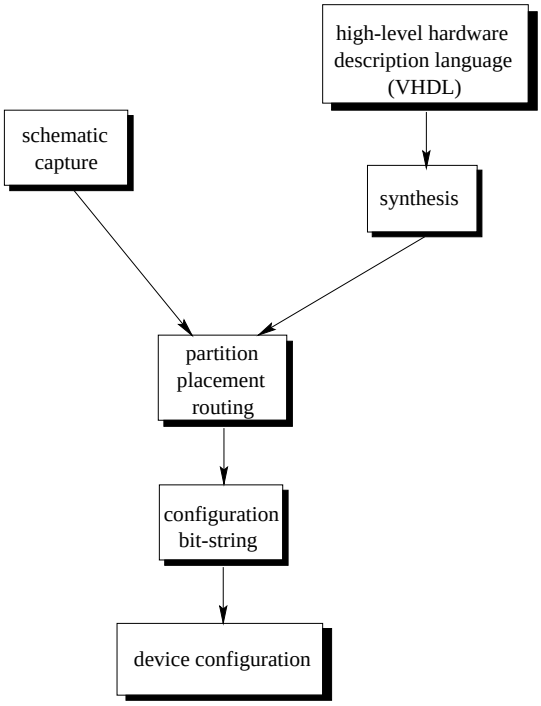


Figure 3.8: Configurable-hardware design flow.

From the architectural point of view, we can distinguish two different types of FPGAs: *fine grain* and *coarse grain* devices. In the former case, the FPGAs are composed of many simple logic blocks (i.e., a logic block may be composed of a single 2-input multiplexer gate), while in the latter case, the FPGAs are composed of fewer, more complex logic blocks (i.e., a logic block may consist of several multiplexers and several memory elements, look-up tables, or even a whole processor).

Although some current commercial FPGAs are coarse grained, and maintain very complex logic blocks, the processing element (PE) of an artificial neural network is not likely to be mapped into a single logic block. Often, a single PE can be mapped into an entire FPGA device. However, current FPGAs offer very high densities: commercial FPGAs offer approximately 250,000 equivalent gates, and, some companies have announced devices with close to 1,000,000 equivalent gates. Figure 3.9 shows the layout of the configurable logic block of the Xilinx XC4000 family (an example of a coarse grained logic block) of FPGAs, which include devices with densities ranging from 2,000 to 250,000 equivalent gates.

Most FPGAs can be reconfigured as often as desired, thereby allowing a single chip to be time-shared by several functions or designs (such operation is also known as run-time reconfiguration). However, the design flow in configurable computing is still several orders of magnitude slower than the compilation of programs, thus limiting the usefulness of reconfigurable hardware.

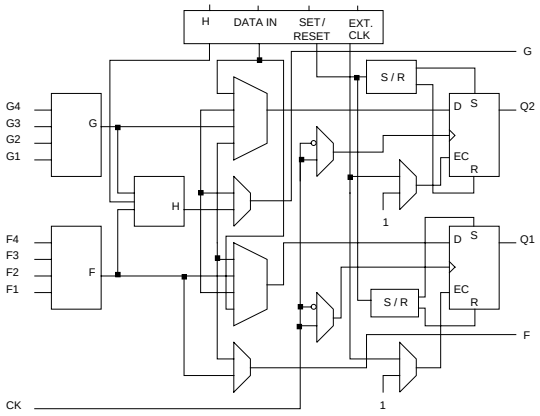


Figure 3.9: Xilinx XC4000 configurable logic block (CLB) architecture.

Artificial neural networks have already been implemented using field programmable devices, initially taking advantage of their potential for rapid prototyping [20, 47] and then of their reconfigurability for density enhancement [55, 76] and acceleration [20]. However, the reprogrammability and reconfigurability of such devices have not been exploited to implement neural networks with in-circuit structure adaptation, the goal of our research.

3.5.2 Rapid prototyping

Field-programmable devices have proven to be very useful for rapid prototyping digital systems and as a low-cost alternative to implement digital systems with limited production runs. The GANGLION project [47], one of the first projects that used FPGAs to implement artificial neural networks, is a clear example of very rapid prototyping of very high-performance neural-based classifier systems.

Artificial neural networks based on field-programmable logic thus offer the advantages of a hardware implementation (namely small size and high-speed operation) coupled with those of rapid prototyping. Although FPGAs do not achieve the power, clock rate, or gate density of custom chips, they provide a speed-up of several orders of magnitude compared to software simulation [81].

Unlike custom ASIC circuits, FPGA devices are shipped fully tested, which implies that users need not deal with issues such as finding test vectors or determining fault coverage. Moreover, FPGA devices can also be used for building *defect-tolerant* computing machines, that is, machines able to operate correctly in the presence of manufacturing errors (using defective FPGA devices and defect-finding algorithms to create a defect database for the compiler), which allows the development of very cheap massively parallel computers [82]. Finally, FPGA devices can be purchased off-the-shelf, eliminating the long delays of ASIC manufacturing.

3.5.3 Density enhancement and acceleration

Reprogrammable circuits enable us to implement separate parts of the same system by time-multiplexing a single FPGA chip through run-time reconfiguration. This approach divides an algorithm into sequential stages; the FPGA device implements one stage at a time (e.g., a backpropagation learning algorithm may be divided into a sequence of feedforward stages and a sequence of backpropagation of the error stages). When the stage's computations are completed, the FPGA is reconfigured for the next stage. This process is repeated until the task is completed. Since only part of the algorithm is implemented at each stage, less hardware is need to implement it and the resulting hardware implementation is cheaper and consumes less power [55]. However, good performance can only be achieved if the reconfiguration time is small compared to computation time.

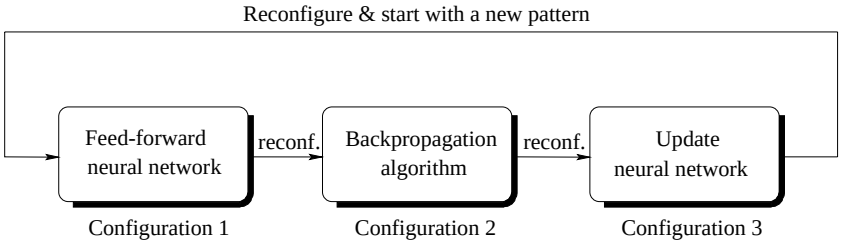


Figure 3.10: Example of run-time reconfiguration: the backpropagation learning algorithm is divided into three stages.

One of the most space-consuming features in digital neural network implementations is the multiplication unit. Therefore, practical implementations try to reduce the number of multipliers and/or reduce the complexity of the multiplier. One typical approach is to use time-division multiplexing (TDM) and a single shared multiplier per neuron. Another approach is to use bit-serial stochastic computing techniques (Section 3.3.2), where stochastically coded pulse sequences allow the implementation of a multiplication of two independent stochastic pulses by a single two-input logic gate. As an example, in [20] a single layer of 12 inputs and 10 outputs of a stochastic neural network was implemented using Xilinx 4003 FPGA. [109] presents an FPGA prototyping implementation of an on-chip backpropagation algorithm using parallel stochastic bit-streams. Another typical approach has been the use of weight and learning parameters values limited to powers of two (thereby reducing multiplications to simple shifts) [46, 132].

3.5.4 Topology adaptation

Digital neural networks with adaptable topologies have been implemented using VLSI techniques [146]. Field-programmable devices besides rapid prototyping and faster operation, enable us to implement artificial neural systems with modifiable topologies. We developed the *Flexible Adaptable-Size Topology* (FAST) neural architecture, an unsupervised learning neural network that dynamically changes its size, which is presented in next chapter. It has been implemented using commercial FPGAs, and has been successfully used in pattern clustering tasks and non-trivial control tasks. This network does

not yet exploit FPGA partial reconfigurability, already possible with FPGA architectures like the Xilinx XC6200 [177, 238]. This feature should allow the development of more complex adaptable-topology neural networks and new hardware implementation approaches.

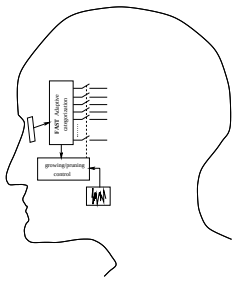
3.6 Summary

Most artificial neural network applications today are executed as conventional software simulations on sequential machines with no dedicated hardware. However, reconfigurable hardware can provide the best features of both programmable and application-specific circuits: high-density and high-performance designs and rapid prototyping and flexibility.

Stochastic neural networks are a promising technique for implementing artificial neural networks: compact neurons can be implemented because of the simplicity of the arithmetic operators and the possibility of an arbitrary resolution of signals. However, the noise sources must be uncorrelated, making the implementation of learning systems very difficult. Online arithmetics appear to be a very promising technique to implement digital artificial neural networks, providing a gain in both performance and in area. Our research, however, is based on the classical positional value representation system. Artificial neural networks have already been implemented using field-programmable devices, initially taking advantage of their potential for rapid prototyping and then of their reconfigurability for density enhancement and acceleration. However, such devices have not been exploited to implement neural networks with in-circuit structure adaptation, which is the main motivation of this research.

Chapter 4

Adaptive Categorization: from Biology to Hardware



"Everything should be made as simple as possible,
but not simpler."

-A. Einstein

From a biological point of view, it has been determined that the genome contains the formation rules that specify the outline of the nervous system. Nevertheless, there is growing evidence that nervous systems follow an environmentally-guided neural circuit building (neural epigenesis) that increases their learning flexibility and eliminates the heavy burden that nativism places on genetic mechanisms [164]. The seminal work of the Nobel laureates D.H. Hubel and T.N. Wiesel on the brain's mechanism of vision [93] describes a prime example of the role of experience in the formation of the neuro-ocular pathways. The nervous system of living organisms thus represents a mixture of the innate and the acquired, the latter precluding the possibility of direct hereditary transmission under Darwinian evolution [193].

Similarly, in the domain of artificial neural networks a special class of learning algorithms has been introduced to deal with the artificial neural network topology problem (Chapter 2) by offering the possibility of dynamically modifying the network's structure and providing *constructive incremental learning*. These techniques appear as an alternative to evolutionary techniques, which have been successfully used for the design and topology-adaptation of artificial neural networks, albeit as the result of an off-line adaptation process.

Learning in artificial neural networks is generally understood as achieving the capability to infer a response to previously unknown situations through the generalization of known situations. Engineers are thus confronted to a trade-off between generalization and robustness [172, 173]: when adding lots of neurons to guarantee fault tolerance, there

is a risk of "learning nothing" if we do not attempt to generalize. From an engineering point of view, "silicon" neurons are not cheap. Therefore, our algorithms definitely need to be cost effective and generate the smallest possible nets. Once we obtain good generalization to a problem, fault tolerance and self-repair can be achieved in many other ways, for example with an embryonics substrate [129].

Engineers are also confronted to the *stability-plasticity* dilemma (how can a learning system preserve what it has previously learned, while continuing to incorporate new knowledge), which has to be solved by any brain system that attempts to makes sense of the flood of environmental signals that relentlessly assails its sensors [72].

A computational approach called Adaptive Resonance Theory (ART) [70, 71] has been proposed to suggest how the brain might solve the stability-plasticity dilemma, how the brain is capable of continual learning as a consequence of its learning stability, how stability implies intentionality, and how this latter implies attention and consciousness [73]. This theory has inspired the development of artificial neural network models with incremental learning capabilities. In particular, there is a family of ART-based computational neural models that *cluster*, *code*, or *categorize* input data in an unsupervised manner.

This chapter presents a structure-adaptable artificial neural network architecture called FAST (for Flexible Adaptable-Size Topology), based on the premises of the ART theory, and particularly motivated by the possibility of a digital implementation using programmable hardware devices. This chapter is organized as follows: Section 4.1 concerns the neural plasticity in the brain, Section 4.2 introduces the concept of adaptive clustering, Section 4.3 describes the Adaptive Resonance Theory, Section 4.4 presents the development of the FAST architecture, Section 4.5 delineates the hardware implementation of the FAST neural network, and, finally, Section 4.6 describes some results and applications including color image segmentation.

4.1 Neural plasticity

When we observe that an infant cannot easily reach a toy, we understand that perception and movement capabilities are not innate, but are progressively developed during a long process of learning.

The genetic program that is carried out during pregnancy provides the children with a set of sensorial organs and a brain ready to use. Within the brain, the 10^{11} neurons are connected into specialized circuits. To know how to treat the sensory information, the brain modifies its structure as a function of its first sensorial experiences: the main structures persist but the particular connections change.

> *"The developing brain can be likened to a highway system that evolves with use: less traveled roads may be abandoned, popular roads broadened and new ones added where they are needed"* [12].

There is growing evidence that the development of the neural circuits of nervous systems depends on the inputs they receive from the environment (neural epigenesis), which increases their learning flexibility. The seminal studies of D.H. Hubel and T.N.

Wiesel on the brain's mechanism of vision [93] are an example of the role of activity and of experience in the formation of the functional properties of neurons and of the refinement of connectivity between neurons.

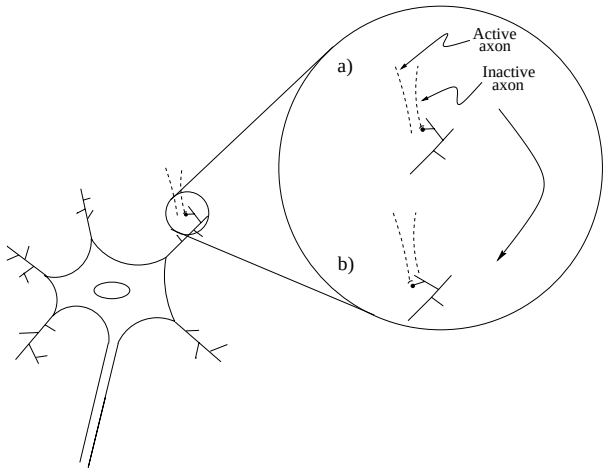


Figure 4.1: Neural plasticity. Besides the adaptation of the neurotransmitters' transmission efficiency in the synapses, neural plasticity entails the formation and elimination of synapses, that is, of connections between neurons. For example, it has been shown that neuronal connections in kittens' eyes reorganize when one eye is artificially sutured and does not receive normal activations from its environment (change from **a)** to **b)**).

These early studies involved suturing closed the lid of one eye in newborn cats. They observed that limiting the activity of one eye interfered with the development of the cat's visual pathways: the axons associated with the closed eye do not segregate into the usual eye-specific brain domains, while the axons associated with the open eye expanded their territory in the visual cortex [105].

Research in humans and primates has shown that the developmental course of the brain involves synaptic over-growth followed by marked selective pruning [45]

Beginning at early stages of fetus development, synaptic density rises at a constant rate, until a peak level is attained (at 2-3 years of age in humans). Then, after a short period of stable synaptic density (until the age of 5 in humans), an elimination process begins, lasting until puberty. Ruppin et al [45] have observed a strong correlation between synaptic density and metabolic energy consumption in the brain, and demonstrated that the performance decrease due to synaptic deletion is small compared to the energy savings. Furthermore, the French neurophysiologist J.-P. Changeux has postulated that "to learn is to eliminate", as opposed to learning by instruction, which implies new growth [44].

It should be noted that an indefinite neural plasticity in certain parts of the brain can be counteractive: this is notably the case of the sensory pathways [105]. As an example, the plasticity of the neuro-ocular pathways in animals is under experience-dependent regulation for a limited period early in the life of animals, known as the *critical period* [66]. In other parts of the brain, a continuing neural plasticity is essential to incorporate new knowledge.

The formation of appropriate synapses is the ultimate step in the construction of brain circuitry and, as brain researcher P. Rakic said, "the synaptic architecture defines the limits of an individual mental capacity" [165]. In this thesis, we are particularly interested in the development of the sensory neural circuits that enable us to learn something about our changing world by means of the capacity of categorization.

4.2 Adaptive categorization

Categorization refers to the process by which distinct entities are treated as equivalent. It is considered as one of the most fundamental cognitive activities because categorization allows us to understand and make predictions about objects and events in our world. Indeed, what we mean by a *concept* is a mental representation of a category [135]. We use concepts to interpret our current experience by classifying it as being of a particular kind, and hence relating it to prior knowledge to provide a means of understanding the world [79]. This is essential in humans, for instance, to be able to handle the constantly changing activation of around 10^8 photo-receptors in each eye [51].

When we learn to sort objects into categories, we note their differences: members of the same category look more alike and members of different categories look more different. This phenomenon of within-category compression and between-category separation in similarity space is called *categorical perception* (CP) [80].

Some psychologists have proposed that *similarity* is an organizing principle. For instance, William James wrote in his book *The Principles of Psychology*: "This *sense of sameness* is the very keel and backbone of our thinking" [101].

However, this theory is somehow controversial given that similarity is usually defined in terms of shared properties, and thus we may see things as similar just because they belong to the same category and not vice versa [135]. Categorization thus touches interesting applied and theoretical issues: for example, a consequence of the similarity view implies that the world is organized for us and that our categories map onto such reality [135].

Geometric models have been among the most frequently-used approaches to analyzing similarity [69]. For instance, the Euclidean distance metric often provides good fits to human similarity judgments. However, to compare things that are richly structured and not just a collection of coordinates or features, it is often most efficient to use hierarchies (parts containing parts) and/or propositions (relational predicates taking arguments) [69].

In our present work, we are interested in two different categorization tasks: the first task is related to image processing, while the second task is related to neurocontrol. In the first task, the process of color categorization is used for image segmentation and recognition. In the second task, an autonomous "intelligent" system self-categorizes (or clusters) its sensor readings in order to integrate the relentless bombardment of signals coming from the environment. It can ultimately serve as a computational model to attempt to explain how we manage to integrate such signals into "unified moments of conscious experience that cohere together despite their diversity" [73], and how a system can achieve continual learning throughout its life.

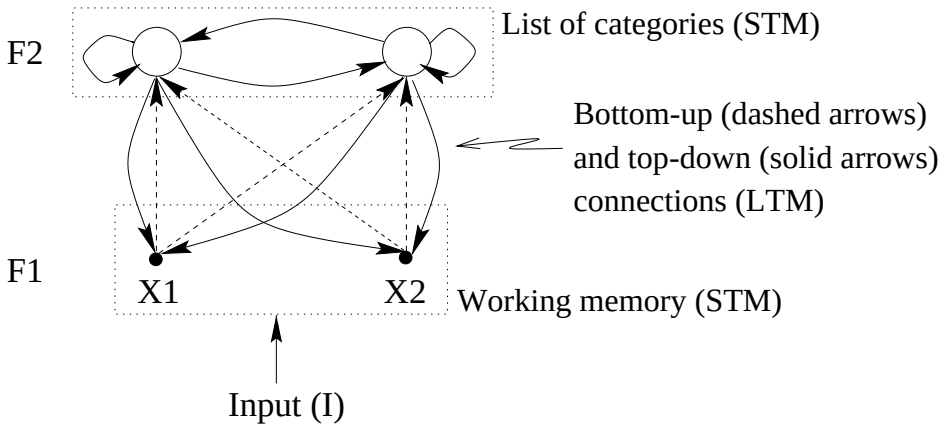


Figure 4.2: ART outline. An ART model is composed of two layers of neurons F_1 and F_2. The activation of neurons in F_1 and F_2 corresponds to a short-term memory (STM) pattern of activation, while the weights on the bottom-up (dashed lines) and top-down (solid lines) connections implement a long-term memory (LTM). Information can flow from the short-term memory to the long-term memory during *learning* (i.e., when then STM patterns and the LTM expectations resonate), and from the long-term memory to the short-term memory during *recall*. Lateral inhibitory and self-excitatory connections of neurons in layer F_2 are essential for competitive learning.

4.3 The Adaptive Resonance Theory (ART)

Self-organizing feature map models [107] and competitive learning models in general [70, 220] combine associative learning and lateral inhibition to enable a system to learn categories. Such models are particularly useful in tasks where the system is given a collection of patterns and is asked to construct a model to explain the observed properties of the patterns. No external supervisor provides the desired outputs or rewards. Such model must reflect the particular statistical structure of the overall collection of input patterns (Section 2.5.3). Nevertheless, under arbitrary environmental conditions, learning becomes unstable [73]. The learned categories may vary with the order of presentation of the input patterns, and the category of a given input pattern may continue to change. One way to avoid this problem is to gradually reduce the learning rate, with a corresponding loss of the network plasticity. Grossberg introduced the Adaptive Resonance Theory (ART) to stabilize the memory of a self-organizing feature map [73].

The ART learning model first assumes the existence of two types of memories: a *short-term memory* (STM) and a *long-term memory* (LTM) [83]. The short-term memory has a limited capacity, but is able to acquire information at very high rates. Conversely, the long-term memory has a huge capacity, but is very sensitive to the rate of arrival of information, and tends to encode information in terms of meaning instead of, for example, sound [19].

An ART learning model consists of a short-term memory and a long-term memory elements. Information can flow from the STM to the LTM during learning, and from the LTM to the STM during recall. An ART neural network is composed of two layers of nodes as shown in Figure 4.2: a *feature representation* layer F_1 and a *category represen-*

tation layer F_2. Weighted connections between the layers in both directions implement a long-term memory. When an input pattern I is presented to the network, a short-term memory activation is generated in the bottom layer F_1, i.e., the *working memory* (the X_1, X_2 pattern of activation in Figure 4.2). The sufficiently active nodes in the bottom layer then emit bottom-up signals to the top layer F_2. Such signals are amplified by the bottom-up interconnection weights (i.e., the long-term memory), and summed before generating a short-term memory pattern of activation across the category representation layer F_2. The LTM in the bottom-up connections thus serve as an adaptive filter, and influence the selection of categories (in the top layer) for the given input pattern I. The categories in layer F_2 activate LTM top-down *expectations* that are matched against the active STM patterns. If this testing process ends in an approximate matching (i.e., the new information can be safely accommodated into existing categories), a fusion of the STM pattern and the top-down expectations occurs, the bottom-up and top-down signal patterns mutually reinforce and converge to a resonant state of short-term memory activation that is incorporated into the long-term memory (i.e., the system learns) [37].

Conversely, if the bottom-up and the top-down information do not match, and a resonance cannot develop, then the F_2 category is quickly reset and a memory search for a better category is initiated. When an uncommitted node at F_2 is selected, the bottom-up and top-down LTM traces learn the STM trace in F_1 and no top-down alteration of the pattern in F_1 occurs. If the full capacity has been exhausted and no adequate match exists, learning is inhibited. The combination of top-down matching, attention focusing, and memory search is what stabilizes self-organizing maps in an arbitrary input environment [73].

To summarize, the ART theory claims that, in order to solve the stability-plasticity dilemma, *only resonant states can drive new learning.* This principle has been used with great success by the artificial neural network community as will be shown in the next section. Grossberg also used his theory of human cognitive information processing to explain several challenging behavioral and brain data including visual perception, visual object recognition, and auditory source identification [73]. Despite the diversity of tasks, the neural circuits that govern such processes seem to rely on a similar set of computational principles, represented in the ART theory. Grossberg ultimately extended his claims to state that the central hypothesis of ART is: *all conscious states are resonant states* [73].

4.3.1 ART-based artificial neural network models

The ART theory has led to an evolving series of artificial neural network models that perform supervised and unsupervised learning [34, 35, 36, 38, 40, 41, 42] (Figure 4.3). In particular, A.E. Alpaydin [6] developed two extensions of ART called Grow and Learn (GAL), and Grow and Represent (GAR), which inspired the Flexible Adaptable-Size Topology (FAST) neural architecture described in this thesis.

ART encompasses a wide variety of artificial neural network models based explicitly on neurophysiology, which perhaps explains why the theory has its own jargon. For example, the current training input pattern is stored in short term memory and cluster seeds (or reference vectors) are stored in a long term memory. The mechanism of

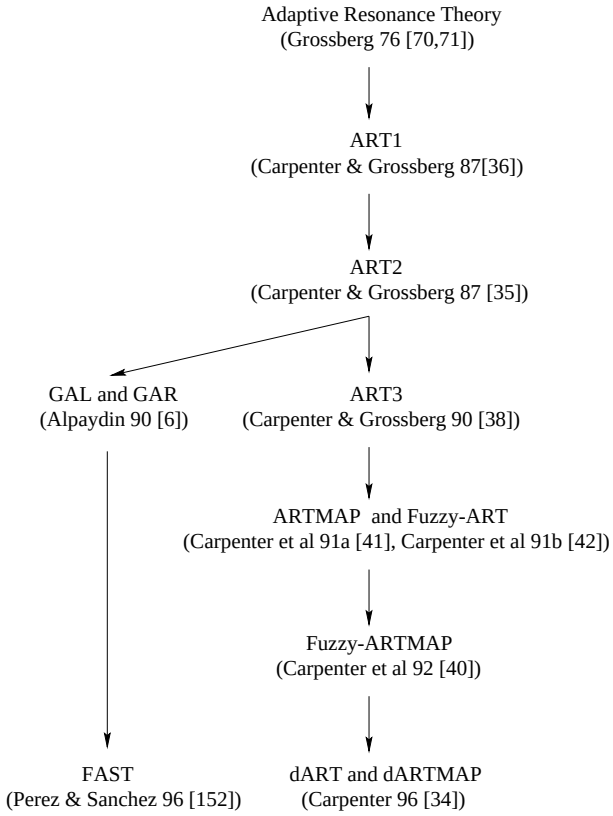


Figure 4.3: Evolution of some ART-based artificial neural network models.

category selection is performed by an *attentional subsystem* and an *orienting subsystem*, the latter performing *hypothesis testing* (the comparison with the vigilance parameter or the threshold). *Stable learning* means that the algorithm converges. So the often-repeated claim that ART algorithms are "capable of rapid stable learning of recognition codes in response to arbitrary sequences of input patterns" [37] merely means that ART algorithms are clustering algorithms that converge, and not, as one might naively assume, that the clusters are insensitive to the sequence in which the training patterns are presented (in fact, quite the opposite is true).

4.3.2 ART and conventional clustering algorithms

ART1 and ART2 were the first computational implementations of the ART theory premises. ART2, for example, is defined algorithmically in terms of detailed differential equations intended as plausible models of biological neurons [35], (even though, in practice, ART networks are implemented using analytical solutions or approximations of these differential equations). The ART1 and ART2 algorithms implement the following steps according to the ART theory:

1. **Potential computation:** The potential or a short-term memory activity (in the representational layer F_1) is computed as a function of the inputs.

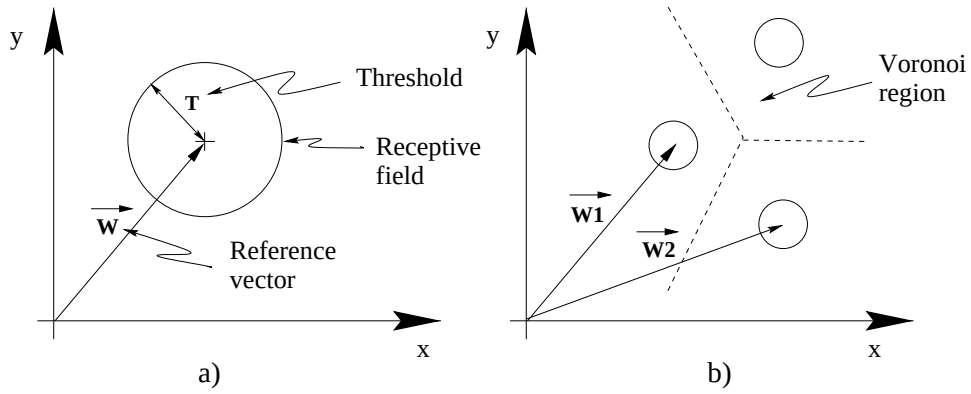


Figure 4.4: (a) Receptive field of a neuron and (b) a particular partition of the input space or Voronoi tessellation (all reference vectors are not shown for clarity)

2. **Bottom-up filtering:** The short-term memory pattern is intensified or diminished by means of a bottom-up adaptive filter.

3. **Category selection:** A short-term memory pattern of activation is generated across the category representation (layer F_2) as a function of the filtered short-term memory pattern in layer F_1.

4. **Matching expectations:** The short-term memory pattern in layer F_2 activates long-term memory top-down expectations which are then matched against the active STM patterns in layer F_1.

5. **Learning:** If the result is an approximate matching, a new STM pattern of activity is computed as a consensus between the STM pattern in layer F_1 and the LTM top-down expectations, and is then incorporated into the LTM traces.

6. **Reset:** If the degree of match between the STM pattern and the LTM pattern (the expectations) is not sufficient, an *orienting subsystem* resets the STM pattern activation in layer F_2 and a memory search for a better category is initiated. An uncommitted node is chosen if the memory is not yet fully used.

From the pure clustering point of view, the distinctive characteristic of the ART models is the use of a *vigilance parameter*, which determines the maximum distance between two vectors in the same category. In terms of pattern clustering it is related to a distance threshold or the cluster diameter [143]. The vigilance parameter is thus used to determine if an input is *sufficiently similar* to those already learned, otherwise activating a new category until the memory capacity is full. The reference vector corresponds to a *prototype* of the associated cluster (category), and determining the *resonant* neurons is equivalent to finding the *nearest* or the *most similar* cluster prototype in the space of input patterns. Sometimes, one may also talk about the reference vector as the center of the *receptive field* or the *sensitive region* of the neuron. A given set of neurons with their reference vectors defines a particular partition of the input space, the so-called *Voronoi tessellation* (Figure 4.4).

0. Start with the set of prototype vectors $P' = \{\}$.
1. Present a new input pattern P and find the closest prototype vector (if any): i.e., find the neuron j that maximizes $\frac{W_j \cdot P}{\beta + ||W_j||}$, where β is a constant value, and $\beta \ll 1$.
2. If $P' = \{\}$ or P is too far from W_j, that is, if $\frac{W_j \cdot P}{\beta + ||W_j||} < \frac{P \cdot P}{\beta + n}$, where n is the length of the binary vectors, create a new neuron j^* with $W_j^* = P$ and make $P' = P' \cup j^*$.
3. If W_j does not match P, that is, if $\frac{W_j \cdot P}{P \cdot P} < T$, where T is the threshold. Then set P'=P'-{i} and go to step 2.
4. If W_j sufficiently matches P, update $W_j : W_j \leftarrow W_j \cap P$.
5. Goto step 1.

Figure 4.5: The inner loop of a clustering ART1 learning algorithm.

As an example of a simplified computational implementation capturing the basic principles of the theory, let's consider an implementation of ART, where each node or neuron j of this implementation of ART maintains a LTM trace that is implemented by a reference vector W_j (a weight vector) and a threshold value T_j (i.e., the vigilance parameter). The self-organization of categories begins by presenting to the network a vector P (*the potential*) that corresponds to a binary vector of length n. As a function of the input vector P the first layer (F_1) generates an input for the competitive stage: a node in the second layer (F_2) receives a measure of similarity between the reference vector W_j and P (i.e., bottom-up filtering). The closest node (i.e., the node with the highest measure of similarity) *wins* (i.e., its corresponding category is selected). The correlation between the input pattern and the reference vector is then used to determine if learning should occur. If the matching is sufficient according to the vigilance parameter, the reference vector W_j is made more similar to the input vector P. Conversely, if the match is not sufficient, an uncommitted unit k (if available) is chosen to code the input pattern P.

In Figure 4.5 we present a simple implementation of the ART1 model in procedural form, an extension which was used to implement an analog clustering ART2 algorithm to work with continuous values [6, 33]. In particular, the extension uses Euclidean distance as a similarity measure and the weight vector adaptation aligns the weight vector W_j closer to the current input vector P using the formula: $W_{ji} = W_{ji} + \alpha * (P_i - W_{ji})$ for all i dimensions of the input space. The clustering implementations of ART have been shown to share a lot of characteristics with classic clustering algorithms such as *k-means* [170].

4.4 GAR: Grow and Represent model

The Grow and Represent (GAR) model was developed by A.E. Alpaydin [6] as an extension of the clustering ART2 neural model. His basic idea was to introduce different and changing threshold values (vigilance parameters) for the various units or neurons in the network. Furthermore, he conceived a pruning mechanism to ameliorate the clustering capabilities of the model. In summary, GAR presents two mechanisms that enhance the ART2 basic clustering: *variable vigilance* and *pruning*.

4.4.1 Variable vigilance

In a clustering ART2 implementation the value of the vigilance parameter T is determined a-priori by the user. A low vigilance leads to broad generalization and abstract prototypes. High vigilance leads to narrow generalization and to prototypes that represent few input exemplars. Learning using ART systems can continue in a stable fashion, with familiar inputs directly accessing their categories and novel inputs triggering adaptive searches for better categories, until the network fully uses its memory capacity [39]. However, the quality of categorization attained by the algorithm is critically dependent on the static vigilance parameter, fixed *a-priori*.

Alpaydin introduced the possibility of having different vigilance parameters (thresholds) and of adapting the active neurons depending on the density of exemplars in every cluster. Basically, each neuron j, when activated, sets its reference vector W_j equal to P, where P is the current input pattern, and initializes its threshold T_j to an initial value T_{ini}. The size of the threshold is then adapted according to the density of exemplars: a so-called *trophy count* τ_j is associated with each neuron. When a neuron j wins a competition, its trophy count is incremented by 1. When τ_j reaches a limit value τ_{max}, the threshold value T_j is decreased (to a minimum value T_{min} defined by the user) so that small sensitivity regions are located in dense regions of the input space, and big sensitivity regions in low density regions. Furthermore, he introduced a modified weight vector update intended to ameliorate the convergence to the mean of a region:

$$W_{ji}(t+1) = W_{ji}(t) + \frac{\alpha * T_j}{1+\tau_j} * (P_i - W_{ji}(t)) \ , \tag{4.1}$$

where α is a step-size constant value. Scaling α with the current threshold T_j assures small changes in small hyper-sphere clusters (i.e., clusters with small threshold values) and big changes in bigger hyper-sphere clusters. The introduction of the trophy count, on the other hand, has the effect that the weight vectors of the units with high trophy values get small changes in their weight values.

4.4.2 Pruning

The pruning in GAR is intended to eliminate the neurons with sensitivity regions that overlap to a great extent with other neuron's sensitivity regions. To achieve this, Alpaydin defined an *awake* mode and a *sleep* mode of functioning. A sleep phase occurs after a certain number of awake phases (it was determined empirically that a sleep phase should

occur every 5 sweeps over the entire training set [6]). During an awake phase the system adapts the weight vectors and the thresholds, and may activate new units if available. During the sleep mode, inputs are presented to the network and trophies are computed. The thresholds are not considered nor modified, and thus each input pattern is simply categorized by finding the nearest neuron. A neuron is eliminated (pruned) if

$$\tau_j < (1-k)\frac{M}{N} \ , \tag{4.2}$$

where M is the number of iterations, N is the number of neurons, and k is a constant between 0 and 1 (typically 0.5). The elimination of neurons with low trophies also allows the system to handle the case when the probability density of the input patterns changes in time.

The GAR extension of ART2 forms clusters of possibly different sizes according to the density of exemplars in a given region of the input space, and allows the system to handle the case when the probability density of the input patterns changes in time. However, it requires that the whole input pattern database, somehow be stored, to iterate through alternate *awake* and *sleep* phases. Furthermore, to consider the trophy count for weight vector update is computationally expensive because it involves a division.

In the following section we present an adaptation of the GAR clustering algorithm which handles these limitations, and carefully considers the possibility of a digital implementation.

4.5 The FAST neural architecture

The Flexible Adaptable-Size Topology neural architecture [152] implements an unsupervised learning neural network that has been developed to handle the problem of *dynamic categorization* or *online clustering* by means of a non-computationally-intensive algorithm.

Categories or clusters are determined by the network itself, based on correlations of the inputs (as in ART). Therefore, given that the quality of categorization attained by such algorithm is critically dependent on the static vigilance parameter, fixed *a-priori*, FAST incorporates Alpaydin's dynamic vigilance parameters and an online *pruning* mechanism based on probabilistic deactivation of neurons. The pruning mechanism complements the inherent *growth* of ART and allows the learning network to adapt to input signals with changing probability densities and to release a neuron when its category is redundant (i.e., it is already represented by another neuron). The resulting learning network and its dynamics are explained in detail in Section 4.5.2.

A number of ART-based learning algorithms and unsupervised artificial neural network models with modifiable structures have already been developed by others [6, 59, 64]. However, most of those algorithms are computationally intensive and it is difficult to adapt them in order to achieve a self-adapting stand-alone digital implementation to be used in real-time control applications. FAST was conceived to be implemented using one or few FPGA devices, whereas other approaches have led to the development of massively parallel neurocomputer architectures [146] and ASICs [49, 95].

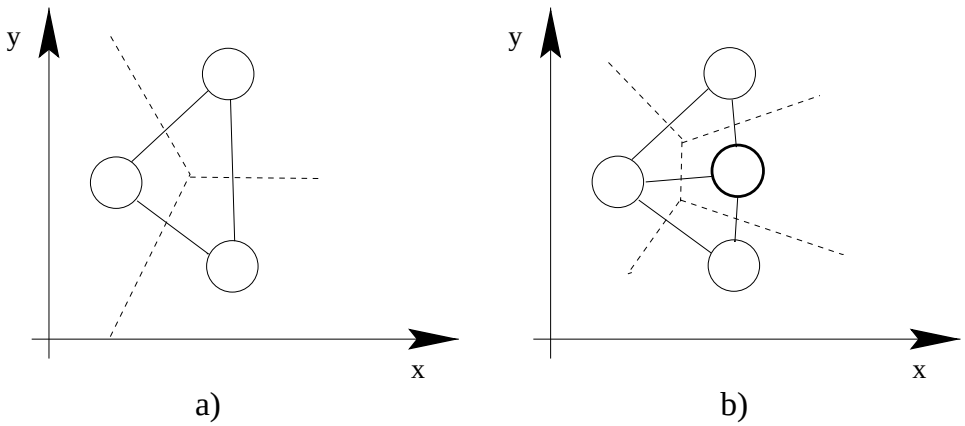


Figure 4.6: Introduction of a new neuron in a Growing Cell Structure. (a) The network starts with a k-dimensional simplex. (b) A new neuron is introduced by establishing connections to the neuron with highest local error and to its neighbors (reference vectors are not shown for clarity).

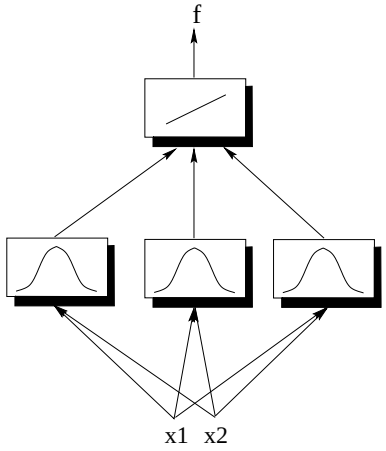
4.5.1 Related work

As we have seen, the FAST architecture is directly related to the seminal work of Carpenter and Grossberg on the ART theory [37], to Alpaydin's extension of ART [6], and similar ontogenic unsupervised algorithms like the Growing Cell Structures [64]. It is also related to similar algorithms developed in parallel using very different sources of inspiration, particularly nonlinear statistics [180]. From the hardware point of view, it is particularly related to Moreno's work [146] and to the Zero Instruction Set Computer (ZISC) chips developed by IBM France [95]. Finally, some hardware implementations of ART networks have been proposed, notably using analog hardware [168, 188].

Growing Cell Structures (GCS)

The Growing Cell Structures algorithm [64] (Figure 4.6) was conceived to alleviate the problem of finding the most efficient topology in a self-organizing feature map [107] in order to obtain adequate mappings when knowledge of the probability distribution of the data is missing. In such cases, the standard topology (a square grid) can lead to poor mappings with rather arbitrary neighborhood relations [64].

Learning in this approach consists of the adaptation of the reference vectors and of the insertion of nodes and connections. The purpose of the growing cell structures model is the generation of a topology-preserving mapping from the input space R^n onto a topological structure of equal or lower dimensionality k. It starts with a k-dimensional structure (a simplex) with its $k+1$ vertices initialized to random vectors in R^n. When an input pattern P is presented to the network, the best matching neuron is determined. This winner unit updates a *local error* value and alters its reference vectors towards the input P, as well as its topological neighbors. After a certain number of steps, a new neuron is introduced by establishing connections to the neuron with highest local error and to its neighbors. As a consequence the accumulated errors of the neurons in the vicinity are reduced.



$$R_i(\vec{x}) = exp\left[-\frac{||\vec{x}-\vec{c}||^2}{\sigma^2}\right] \ ,$$

where $\vec{x}$ is the input vector
and $\vec{c}$ is the reference vector.

Figure 4.7: A network of localized receptive fields or Radial Basis Function neural network (RBF) [163] and its Gaussian transfer function.

In contrast to other approaches all parameters are constant, eliminating the need to define a decay (or annealing) schedule. However, due to the irregular and dynamic graph structure used, the GCS model is more difficult to implement than models with a predefined regular structure.

Resource Allocating Networks

Moody and Darken [141] and then Platt [161] have used networks whose transfer function is a Gaussian (Figure 4.7), inspired by approximation theory. Basically, the transfer functions define *receptive fields* in the input space of the system. When a new input pattern is presented to the system, it activates a certain receptive filter, which then updates its parameters (learns) locally. While global learning systems (i.e., a feedforward network with backpropagation adaptation) can achieve greater generalization from each training pattern (due to the global adaptation), a localized receptive field approach is computationally more efficient if data is not expensive to obtain [140]. It has also been found that similar structures are present in real nervous systems [93].

In [161], Platt presented his Resource-Allocating Network (RAN). This network consists of two layers: the first layer is composed of neurons with Gaussians as transfer functions, while the second aggregates outputs to generate the function that approximates the input-output mapping over the entire space. The algorithm automatically sets the number of Gaussian neurons and adjusts the center of the Gaussian functions based on the error at the output. The result is an algorithm that merges unsupervised and supervised learning and learns much faster than the conventional Backpropagation algorithm.

More recently, Schaal and Atkeson [180] presented an extension of the RAN network based on *non-parametric statistics*, called Receptive Field Weighted Regression (RFWR). This system is intended to achieve robust incremental learning without competitive learning. The network adapts the number of receptive filters based on a vigilance

parameter (or threshold value) as in ART, adapts the size of the receptive filters by gradient descent, and prunes a receptive field if it *overlaps too much* with another receptive field. This overlap is detected when an input vector activates two receptive fields beyond a given threshold. The model seems very promising and introduces interesting ideas for the development of ontogenic unsupervised systems. However, the system is computationally intensive, preventing a direct digital implementation. Our system exhibits similar characteristics obtained with a simpler approach, even though it may be less robust.

A VLSI architecture for evolvable neural models

In [146], Moreno developed a VLSI neurocomputer architecture for certain structure adaptable algorithms, so-called *Region of Influence* or ROI incremental algorithms [144], (also improperly called *evolvable artificial neural networks* [84, 146, 155]) such as Restricted Coulomb Energy (RCE) [167], Grow and Learn (GAL) [6], Incremental Radial Basis Functions [161], and the Probabilistic neural networks (PNN) [196]. The architecture consists of a dynamic ring array of RISC processors (Chapter 2). Each processor contains a pipeline, a three-port register file of 128 16-bit registers to store the weights and training patterns, an ALU composed of a multiplier, an adder/substractor, a boolean unit and a threshold function, and an I/O block.

Zero Instruction Set Computer (ZISC)

The Zero Instruction Set Computer (ZISC) system [49] is a fully-integrated digital implementation of the Restricted Coulomb Energy (RCE) algorithm [167] and the K-nearest Neighbor (KNN) algorithm. The ZISC036 [95] is the first commercial chip of the product family. It contains 36 neurons, each containing a register file to store the reference vector and a distance evaluation unit to ensure a high level of parallelism. Up to 16,382 different categories can be defined in the network. The system is capable of identifying an input vector pattern within a 1- to 64-dimensional space, with 8-bit precision. The similarity measure can be an *L1 norm* (i.e., Manhattan distance) or an *LSUP norm* defining a hyper-polyhedral or a hyper-cube influence field respectively (Figure 4.8).

4.5.2 FAST learning algorithm

Each FAST neuron j maintains an n-dimensional reference vector W_j, and a threshold, T_j, determining its *sensitivity region* (or receptive field), i.e., the input vectors to which it is "sensitive" (Figure 4.9). At the outset, no neurons or categories are activated in the network. Input patterns are then presented and the network adapts through application of the FAST algorithm (Figure 4.10), which is driven by three processes: learning, incremental growth, and pruning.

1. *Learning.* The learning mechanism adapts the neuronal reference vectors; as each input vector P is presented to the network, the distance $D(P, W_j)$, between P and every reference vector W_j, is computed. If $D(P, W_j) < T_j$ (the threshold of neuron j), W_j is updated as follows:

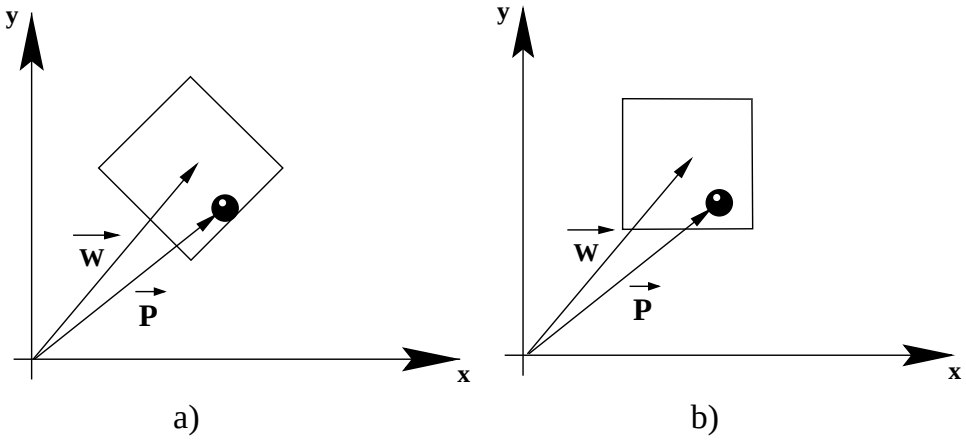


Figure 4.8: (a) Receptive field defined by a L1 norm where $d = \sum_i |P_i - W_i|$ and (b) by a LSUP norm where $d = max_i(|P_i - W_i|)$. $\vec{W}$ is the reference vector of the receptive field and $\vec{P}$ is an input vector.

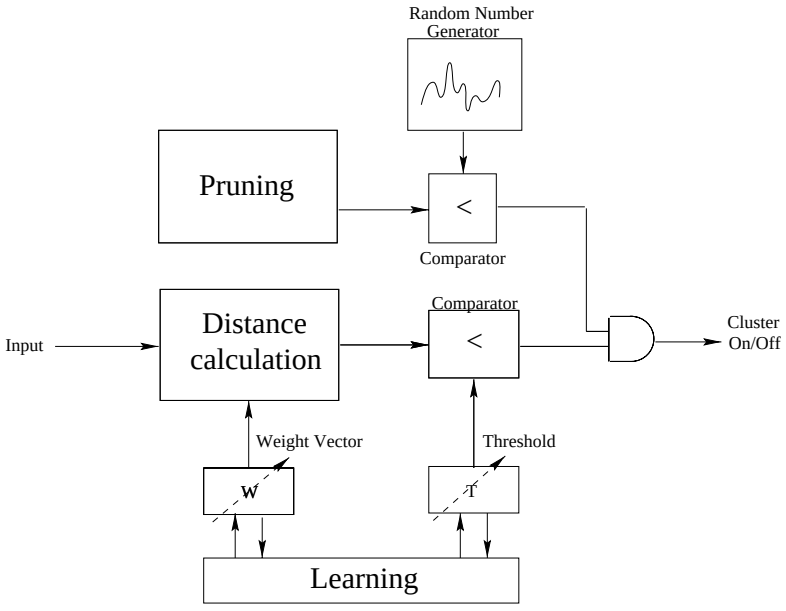


Figure 4.9: Block Diagram of a FAST neuron. A FAST neuron contains a distance calculation block (Manhattan distance), a reference vector and threshold adaptation block (learning), and a deletion block (pruning). The weight vector W and the threshold T determine the sensitivity region of the neuron. An operational neuron is activated if the Manhattan distance between its reference vector and the current input vector is less than the threshold, and may be deactivated by the pruning block.

0. Start with the set of prototype vectors $P' = \{\}$.

1. Present an input pattern P and compute $D(P, W_j)$ for every operational neuron j. $D(P, W_j) = \sum_{i=1}^{n} \mid P_i - W_{ji} \mid$.

2. If $D(P, W_j) > T_j$ for all j, activate a new neuron j^* by initializing its reference vector to $W_j^* = P$, its threshold to $T_j^* = T_{ini}$, and its deactivation parameter to $Pr_j^* = 1$.

3. If $D(P, W_j) < T_j$, where j is an active neuron, and T_j is its threshold, update W_j and T_j as follows:

$$W_{ji}(t+1) = W_{ji}(t) + \alpha T_j(P_i - W_{ji}(t)),$$

$$T_j(t+1) = T_j(t) - \gamma(T_j(t) - T_{min}),$$

where α, γ, and T_{min} are learning constants. Update the global T_{ini} parameter to $T_{ini} = T_{ini}(t+1) - \gamma(T_{ini}(t) - T_{min})$.

4. If several neurons are activated in step 3, deactivate one of the neurons if $rnd() > Pr_j$, where Pr_j is the deactivation parameter, and $rnd()$ is a uniformly distributed random number in the range $(0, 1)$. Increase the activated neurons' probability of deactivation as follows: $Pr_j(t+1) = Pr_j(t) - \eta(Pr_j(t) - Pr_{min})$, where η and Pr_{min} are pruning constants.

5. Goto step 1.

Figure 4.10: The inner loop of the FAST learning algorithm.

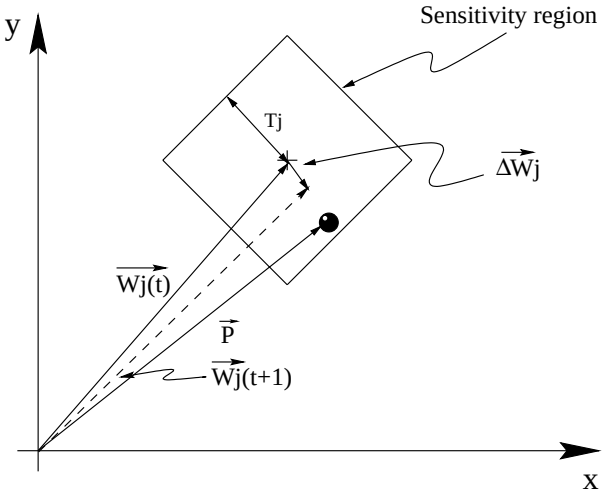


Figure 4.11: FAST learning. When an input vector pattern $\vec{P}$ lies within the sensitivity region of neuron j, it adapts its reference vector $\vec{W}_j$ and its threshold value T_j.

$$W_{ji}(t+1) = W_{ji}(t) + \alpha * T_j * (P_i - W_{ji}(t)) \ , \tag{4.3}$$

where α is a learning parameter in the range $(0, 1]$ (Figure 4.11). In our implementation, the Manhattan distance $D(P, W_j)$ is used as a measure of similarity between the reference vectors and the current n-dimensional input:

$$D(P, W_j) = \sum_{i=1}^{n} \mid P_i - W_{ji} \mid \tag{4.4}$$

This distance gives rise to the diamond-shaped sensitivity region of neuron j shown in Figure 4.11. The activation of neuron j also entails an exponential decrease in its threshold value:

$$T_j(t+1) = T_j(t) - \gamma(T_j(t) - T_{min}) \ , \tag{4.5}$$

where γ is a gain parameter in the range $(0, 1]$, and T_{min} is the minimal threshold. This threshold modification decreases the size of the sensitivity region for neurons in high-density regions of the vector space (Figure 4.12).

2. *Incremental growth.* When an input vector P lies outside the sensitivity regions of all currently operational neurons, a new neuron is added (Figure 4.13), with its reference vector set to P and its threshold set to an initial value T_{ini}. The value of T_{ini} should also be decreased in order to avoid having sensitivity regions contained within other sensitivity regions after successive deactivation-activation phases, a situation which causes instability. The mechanism we introduce to solve this problem consisted on randomly taking a T_j as the updated value of T_{ini}, since stability can be guaranteed simply by maintaining T_{ini} near the T_j mean.

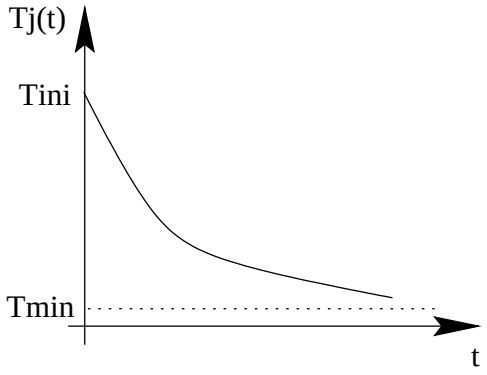


Figure 4.12: Evolution of the threshold value in time.

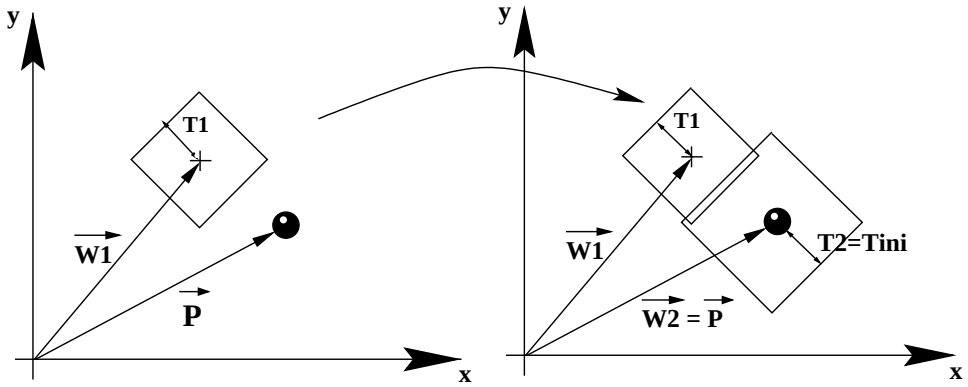


Figure 4.13: Growing in FAST. An ART-based growing mechanism activates a new neuron when a sufficiently different input ($\vec{P}$) is found (i.e, when the distance between the input vector and the reference vector ($\vec{W}_i$) is larger than its threshold T_i).

3. *Pruning.* The network decreases in size (i.e. output neurons are deleted), through a pruning process (Figure 4.14). The probability of an operational neuron being deleted, Pr_j, increases in direct proportion to the overlap between its sensitivity region and the regions of its neighbors. The overlap between the sensitivity regions of several neurons is estimated by computing the frequency of activation of the overlapping neurons with the same input vector. Therefore, the deletion probability of a neuron increases linearly towards a maximum $1 - Pr_{min}$ every time an input vector activates two or more neurons:

$$Pr_j(t+1) = Pr_j(t) - \eta(Pr_j(t) - Pr_{min}) \ , \tag{4.6}$$

where η is a constant that determines the decay of the probability parameter Pr_j. Once a deletion has occurred the probability Pr_j of the units involved is reset.

4.5.3 FAST dynamics

In the last section we have described the learning, growing, and pruning mechanisms of FAST. This section provides a description of the FAST clustering mechanisms, and

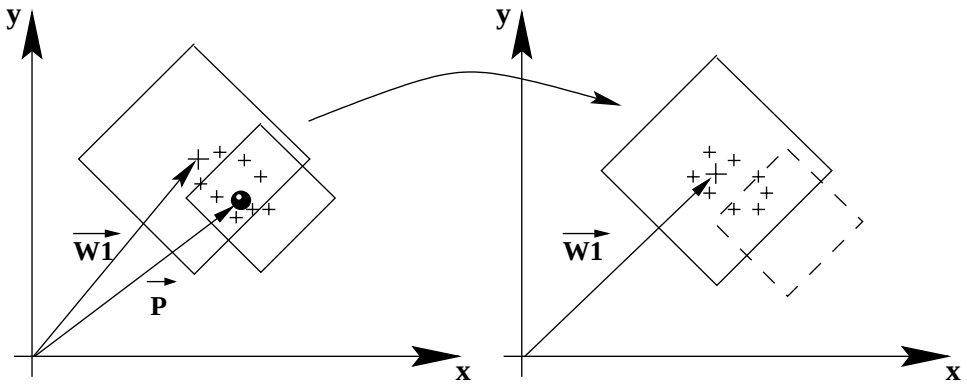


Figure 4.14: Pruning in FAST. The pruning mechanism deactivates a neuron (the dashed rhombus) with a certain probability according to the overlapping of the sensibility regions of the neurons (i.e., the rhombi).

particularly of their dynamics.

Although many other clustering algorithms do exist, they generally need to access the entire database to determine the cluster centers by statistical means. FAST dynamically adapts the centers and sizes of the receptive fields. Nevertheless, it remains quite sensitive to the order of presentation of the input vector patterns.

To further describe the dynamics of the learning algorithm, let us begin by noting that we did not mention a *winner-take-all* operator in the last section. Indeed, this algorithm allows a neuron to adapt its reference vector if it is *sufficiently similar* to the input vector pattern (i.e., the input pattern lies within the corresponding sensitivity region) even if it is not necessarily the nearest neuron. In Figure 4.15 we show how a single input vector may force the adaptation of two different neurons.

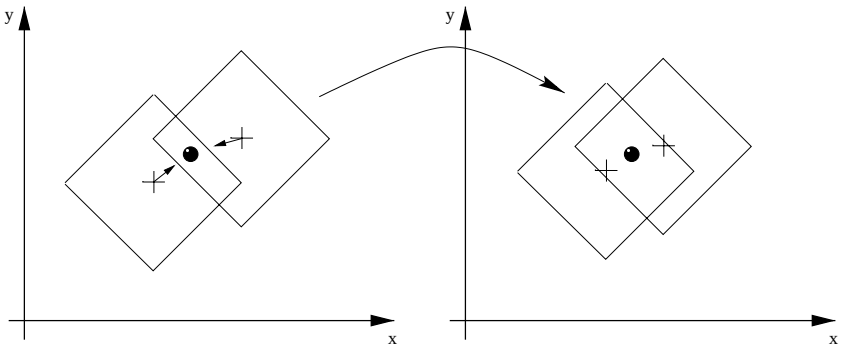


Figure 4.15: FAST neurons adaptation.

This process does not interfere with the competitive learning principles, since the objective remains a density-based clustering. However, a problem may appear when depending on the order of presentation of the input patterns, two or more different neurons may overlap *too much* (which is also possible with other algorithms that use a winner-take-all operator [180]). The pruning mechanism (briefly described in last section) is an attempt to solve this problem by deactivating probabilistically one of the neurons when multiple overlapping sensitivity regions are detected.

The winner-take-all operator is considered an essential mechanism to competitive learning systems [106]. On the one hand it is essential for self-organization, and on the other hand the winner-take-all is essential to produce a single response as a result of the self-organization. Nonetheless, our approach, without winner-take-all but with probabilistic pruning, has been shown to work similarly in spite of possible oscillations in the activations of certain neurons depending on the order of presentations of the input vectors. As an example, we will describe in the following sections a color segmentation application where, given the RGB components of a pixel of a color image, the neural network activates a single color cluster as the output of the system.

Further discussion on this unusual clustering algorithm can be found in Section 5.8 where a cerebellar model is described. In such an algorithm, a winner-take-all is not needed to determine the output of the system. Instead, a pondered sum of activations is used to generate the output of the system. The interest of not having a winner-take-all operator is to provide a self-organizing algorithm by means of local computation, at least for simple density distributions of the input patterns, thus enabling a simpler digital implementation of the system. For more general and biologically plausible approaches, see the work on self-organization of topographic receptive fields and lateral interaction by Sirosh and Miikkulainen [194].

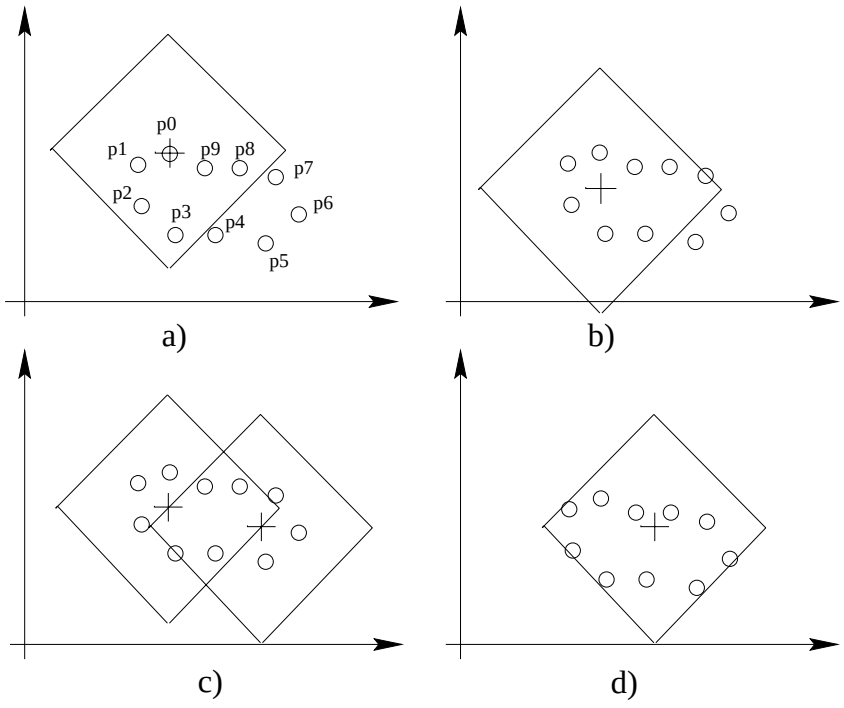


Figure 4.16: Example of FAST dynamics. Given a single cluster of input vector patterns like the one in a), the network activates a first neuron, b) learns, c) grows, and d) probabilistically prunes neurons with overlapping sensitivity regions.

Example of FAST dynamic clustering

To illustrate a typical case of learning, growing, and pruning, let us suppose that we have a single cluster of points like the one in Figure 4.16a. At the outset, there are no neurons activated in the network. When a first input vector of the cluster is presented to the network, a first neuron is activated by setting its reference vector W_j equal to the input vector pattern p_1. After other input patterns p_0, p_2, p_3, or p_9 are randomly presented, the reference vector W_j (i.e., the center of the cluster) will approximately correspond to the center of the presented cluster of points according to their density (Figure 4.16b). If, occasionally, one of the patterns p_4, p_5, p_6, or p_7 is presented to the network, a new neuron is activated as shown in Figure 4.16c, which adapts its center as if the left points p_4, p_5, p_6, p_7, and p_8 constitutes a separated cluster. However, every time one of the shared input vector patterns p_3, p_4, p_8, or p_9 is presented to the network, the probability of deactivating one of the two neurons increases and will possibly leave a single neuron covering the entire cluster (Figure 4.16d).

The order of presentation of the patterns is thus very crucial. Another typical case is the following: suppose that the probabilistic pruning deactivates the neuron to the right in Figure 4.16c, and that one of the input patterns p_4, p_5, p_6, or p_7 is then presented to the network. The algorithm will re-activate a new second neuron, and the clustering will be similar to what it was before the pruning. In certain tasks where the density distributions of the input patterns is variable, FAST can provide an approximate clustering solution. This will be described in more detail in Chapter 6.

Fault-tolerance

An incremental learning algorithm like the one implemented by FAST is inherently fault-tolerant, at least if sufficient units are available. To illustrate this fact, in Figure 4.17a we present three FAST neurons associated to three separate clusters of input vector points. If we suppose that a fault occurs in neuron 2 and that a fault-detection mechanism is able to isolate the neuron from the system (i.e., by using an *embryonic* substrate [129]), when one of the input vector patterns of cluster 2 (for example vector $\vec{a}$) is presented, a new neuron (i.e., neuron 4 in Figure 4.17b) is activated for that given cluster and learning continues as already described.

4.5.4 FAST digital implementation

Recent advances in semiconductor devices have enabled us to use programmable hardware devices such as Field-Programmable Gate Arrays (FPGA) [212] and Field-Programmable Interconnection Circuits (FPIC) [94] which allow users to reconfigure their internal circuit connections and node logic functionalities (Chapter 3). Such devices afford rapid prototyping and relatively inexpensive neural network implementations. The reprogrammability of field-programmable hardware devices enables the implementation of artificial neural networks with in-circuit structure adaptation.

Our first prototype of the FAST neural network architecture was implemented on a custom machine called LOPIOM (Figure 4.18), designed in our laboratory and based on programmable logic [147]. It is composed of a 68331 microcontroller, four Xilinx

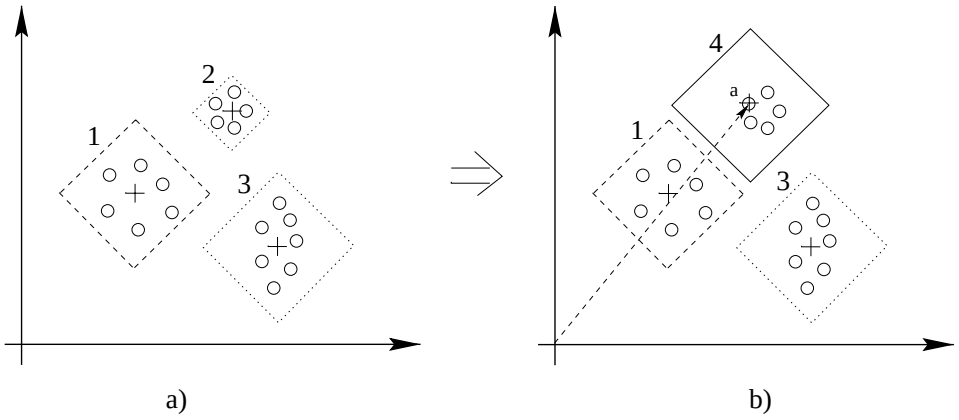


Figure 4.17: Fault-tolerance in FAST.

XC4013-6 FPGA chips (each with approximately 13,000 equivalent gates [237]), and four FPID (Field Programmable Interconnection Devices) chips (Crossbar switches I-Cube 320, each with 320 programmable pads [94]). We used two of the Xilinx XC4013 FPGAs and two of the I-Cube 320 FPIDs to implement four FAST neurons. The sequencer of the network was implemented in one of the Xilinx XC4013 chips along with two of the FAST neurons. The other two FAST neurons are implemented in the second XC4013 chip along with a bank of I/O mapping registers.

To date, two such FAST networks have been implemented: one for two-dimensional input vectors and another for three-dimensional input vectors. In both implementations we used 96% to 98% of the Configurable Logic Blocks (CLBs) of the chip. However, only 66% to 79% of the CLBs were actually used to implement logic circuits, while the rest were used for routing. In the 2D-input network we used 30% of the flip-flops, while the 3D-input required up to 35% of the memory elements. The sequencer, the FAST neuron architecture, and the I/O mapping register bank will be briefly described below.

The Fast neuron

A FAST neuron is composed of three blocks: Manhattan distance computation, learning (i.e., modification of reference vectors and thresholds), and pruning (Figure 4.19). The system currently supports 8-bit computation. Each neuron includes nine 8-bit adders in the 2D case and twelve 8-bit adders in the 3D case, and a single 8-bit shift-add multiplier, so that additions occur in parallel but the multiplications involved in the learning phase are executed sequentially. The maximal number of multiplications during the learning phase is five for the 2D case and six for the 3D case, and each requires 8 clock cycles.

To implement the pruning process, each neuron includes a random number generator (Figure 4.20), consisting of an 8-cell heterogeneous, one-dimensional cellular automaton [90, 91]. It has a length cycle of 255 and is implemented using a small number of components: 8 D flip-flops, 5 two-input XOR gates, and 3 three-input XOR gates. Each cell implements one of two possible ways: the next state of the current cell is an XOR function of the two neighbor cells (*rule 90* according to Wolfram's scheme of classification of cellular automata rules [236])

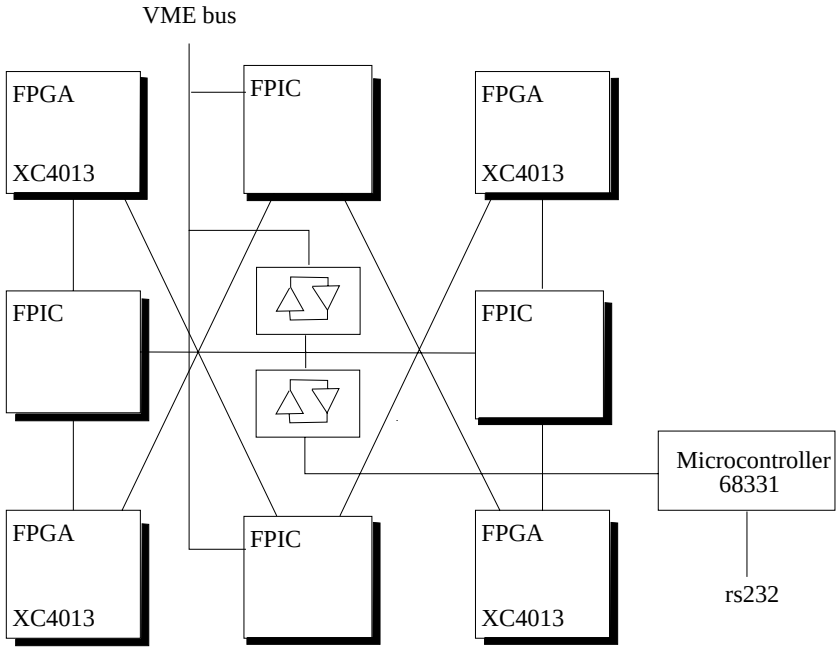


Figure 4.18: Lopiom board architecture [147].

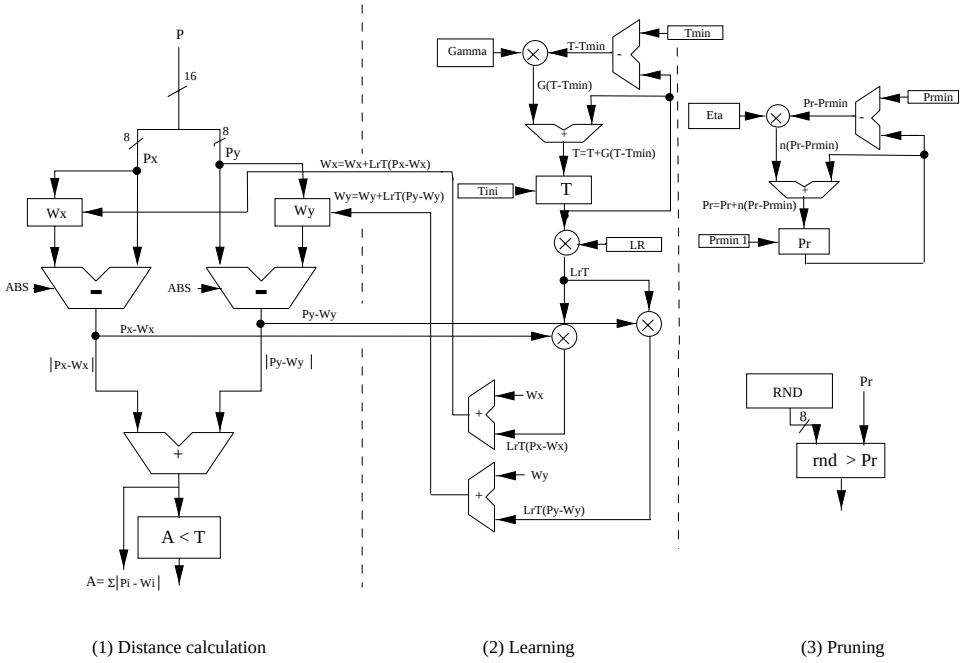


Figure 4.19: Architecture of a 2D-Input FAST neuron. The first block implements the Manhattan distance calculation with 3 adders. A comparator is used to determine the activation of the neuron. The second block contains four adders and a single serial multiplier for the reference vector and threshold adaptation. The pruning block uses the same multiplier of the learning block and contains two additional adders, a random number generator, and a comparator.

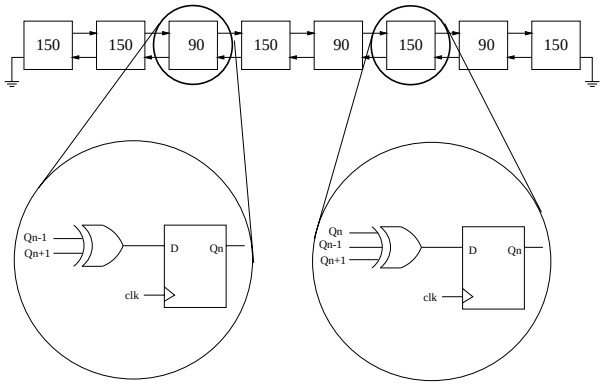


Figure 4.20: Random number generator based on a 1D heterogeneous cellular automaton.

or the next state of the current cell is an XOR function of the two neighbor cells and its current state (*rule 150* according to Wolfram's scheme of classification of cellular automata rules [236]). During startup of the system, the random number generator is set to "11111111". Finally, each neuron maintains an *activation register* that indicates whether it is active or not.

The sequencer
The sequencer is a finite state machine (FSM) which handles the addition and deletion of neuronal units in the output layer and synchronizes the neuron's operation during Manhattan distance calculations, weight and threshold updates (learning), and probabilistic unit deactivation (pruning).

I/O mapping register bank
The hardware device is connected to a host computer which is used to generate input vectors for the system and to display its outputs. The host computer communicates serially with the 68331 microcontroller that reads and writes signals from and to the I/O mapping registers. The I/O mapping register bank consists of ten 8-bit registers used to map every input and output of the neural network. A polling subroutine runs on the 68331 microcontroller in order to generate the read/write control signals and to receive or send data when the host computer requests it. A VME bus connection is also available on the board, but it has not been tested to date.

It should be noted that the process of partition, placement, and routing in the design flow of FPGA devices is not deterministic, and thus the performance and area requirements of a given design may change between different runs. In the following, we present the performance and area results of a typical run of our system.

The maximal XC4013-6's pin-to-pin delay of the 2D-input FAST network is around 120 ns, and the I-CUBE's pin-to-pin delay is 12 ns. The processing time per input vector is up to 57 clock cycles depending on the number of multiplications during learning (Figure 4.21), thus enabling the introduction of a new input vector approximately every 7.5μs. For the 3D-input case, the maximal pin-to-pin delay in the XC4013-6's is around

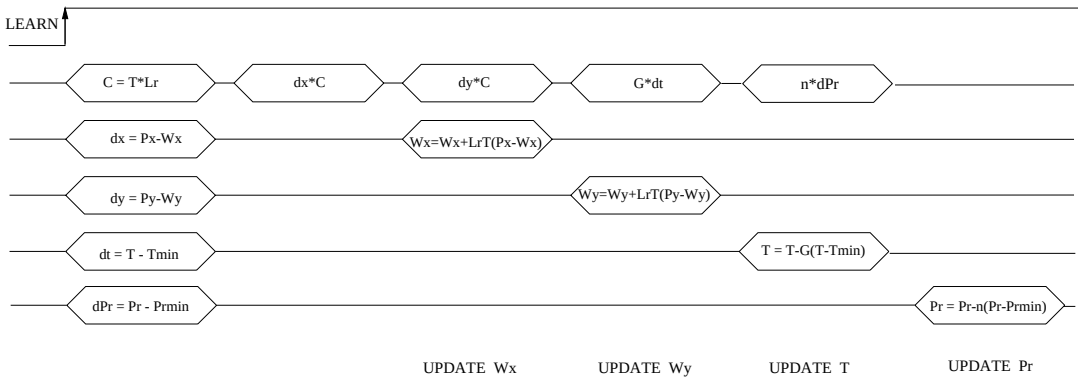


Figure 4.21: Timing sequence of operations in a 2D input FAST network. A single multiplier is time multiplexed during the learning phase. The names of the operands correspond to those of Figure 4.19.

173 ns and a maximum of 66 clock cycles is needed for a learning step, thus enabling the introduction of a new input vector approximately every 8.2μs. See Figure 4.21 for the details on the sequence of operations during the FAST processing of an input vector. Our tests used a slower input rate due to the serial link, but future applications will exploit faster interfaces.

4.5.5 Results and applications

Unsupervised neural networks have been widely used in clustering tasks, feature extraction, data dimensionality reduction, data mining (data organization for exploring and search), information extraction, density approximation, data compression, etc.

To evaluate FAST, two different applications were chosen: (1) a color image segmentation problem, and (2) the inverted pendulum problem. The first application evaluates the clustering properties of the algorithm, allows a comparison with a fuzzy Kohonen clustering algorithm, and allows the test of the hardware implementation. Furthermore, it evaluates the real-time learning and size adaptation potential of the network. The inverted pendulum problem evaluates the on-line learning capabilities of the algorithm as well as its clustering ability. In this second application, our system demonstrates a faster approach to solving a non-trivial control task. A detailed description of the second application and an extension of the architecture are presented in the next chapters.

4.5.5.1 Color image segmentation

The FAST algorithm has been applied to a color learning problem. Given a digital image from a real scene, the problem is to cluster image pixels by chromatic similarity properties. Pixels of similar color belong to a so-called *chromatic class*. By identifying chromatic classes in an image and assigning a code to each class, we can perform color image segmentation. For example, an orange sphere with small bumps on its surface can be quickly recognized as an orange fruit.

Some of the problems arising with the use of color require that a proper color representation, amenable to computation, be determined. In our experiments we consider

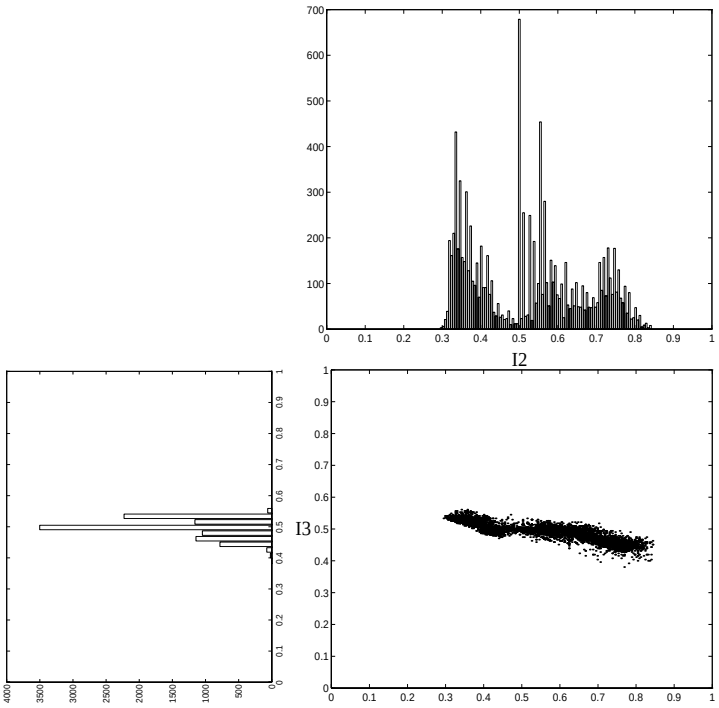


Figure 4.22: Distribution of an image's chromatic data in I2-I3 space (bottom right) and the corresponding chromatic coordinate histograms, i.e., the frequency distribution of the data, along the I2 (upper right) and I3 (bottom left) axes.

the RGB (red, green, blue) components of a color image. Thus, at the outset, every pixel is represented in a 3D space. Then, a color-space transformation is applied so as to represent the pixels in a 2D space called I2-I3 [149]. This 2D space is more suitable for color representation than the RGB system because in the latter, all three variables are dependent on the light intensity. Coordinates in the I2-I3 system appear to be more correlated than in the RGB system [26, 120].
The transformation is as follows:

$$I2 = R - B \tag{4.7}$$
$$I3 = G - 0.5(R + B) \tag{4.8}$$

These bi-dimensional pixel coordinates (I2,I3) are randomly presented to the FAST neural network, which determines clusters in the I2-I3 space, corresponding to color clusters in the RGB space. The resulting color clusters can then be used to construct a segmented image from the original image for analysis.

- **Performance gain.** As have been shown, for a particular partition-placement-routing run, our system is able to handle the introduction of a new input vector approximately every $8.2\mu s$.

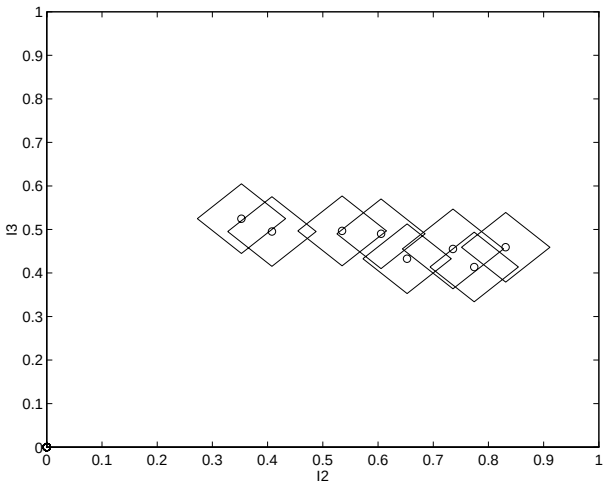


Figure 4.23: Resulting I2-I3 chromatic clusters. The diamond-shaped regions represent clusters in the I2-I3 space corresponding to color clusters in the image, and the circles represent the centers of the clusters.

In [120], neural and statistical methods for adaptive color segmentation were compared. In these tests, a set of 400x400 pixel images were used, and a maximum of 30 classes were allowed. The processing of one of such image running on an IBM RISC 350 station took 30 seconds. If an extension of our 3D-input system were used (for clustering directly in RGB space) with these images, it would take 2x($8.2\mu s$x400x400) = 2.62 seconds (one phase is needed for clustering and a second phase to generate the segmented image): our system is about one order of magnitude faster. One possible extension is to use denser Xilinx devices, such as the XC4036 devices, rather than the older XC4013 devices we have been using until now.

- **Test 1.** The hardware implementation was tested with the I2-I3 points shown in figure 4.22, which correspond to an image of 445x494 pixels (the histograms in Fig 4.22 show their density distribution), but the lack of sufficient neurons has led to poor results. We therefore present below the results of a larger, simulated system. Figure 4.23 shows eight diamond-shaped chromatic clusters identified by the network after presentation of several hundred input vectors. We have compared these results with the reference vectors obtained by another neural network learning algorithm which combines the thresholding in I2-I3 space and the fuzzy Kohonen clustering approach [26]. This latter algorithm generates eight ellipsoidal chromatic clusters, after presenting all the images' color information in the I2-I3 space, while our algorithm determines clusters in a dynamic manner. Furthermore, we found that the eight chromatic clusters obtained by FAST are a good estimation of those obtained by the non-dynamic model, i.e., the eight FAST clusters lie very close to the thresholding-fuzzy Kohonen clusters.

- **Test 2.** A second experiment was issued to test the hardware implementation. We considered a 294x353 pixels, 61-color image of Van Gogh's *Sunflowers*. We

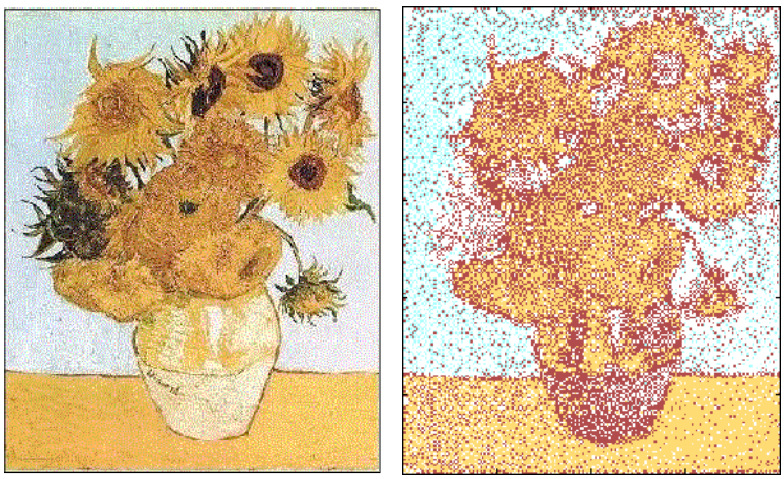


Figure 4.24: (a) Van Gogh's *Sunflowers* in a 61-color image, and (b) FAST-segmented image. The objects in the image are consequently easily recognized by image analysis algorithms.

randomly took a pixel from the image and presented its corresponding color information in the I2-I3 space to the hardware implementation of the FAST neural architecture. The resulting segmented image after 10,000 input vector patterns is shown in Figure 4.24 alongside the original image. Four chromatic clusters were identified (one per neuron), and objects in the image are consequently easily recognized by image analysis algorithms, i.e., the vase, the flowers, the table, and the wall.

4.5.5.2 Neurocontroller adaptation

A major problem in nonlinear control is the tuning and adaptation of the controller. For this purpose, a model of the process is usually developed. Then, following an approximation of the inverse relation between the desired outputs and the control actions, the controller is adjusted. However, for many real-world problems there is no available quantitative data regarding input-output relations, rendering analytical modeling very difficult [27]. Furthermore, errors in the model can lead to poor performance of the controller. Artificial neural networks have been proposed as an alternative for implementing adaptive controllers which are capable of learning [138, 226]. Nevertheless, determining the network's topology is a difficult problem, limiting the usefulness of this approach. A straightforward solution is the use of an *ontogenic* neural network that adapts to the environment online. We have used the FAST unsupervised clustering network for adapting the structure of a reinforcement-based neurocontroller. An empirical evaluation of the resulting controller shows that it surpasses other reinforcement learning methods [156, 157]. In the following chapters we present a detailed description of the system and an extension of the neurocontroller, utilized for autonomous robot navigation.

4.6 Spatiotemporal FAST clustering

The unsupervised learning paradigm has been used mostly to cluster or categorize multidimensional input vectors in the space. However, Grossberg and his colleagues have explored the use of the ART theory to explain many auditory phenomena [72], that is, how the brain is capable of adapting its categorizing of signals in time. They have developed a model based on the idea that the ear's mechanical and neurophysiological filters divide sounds into groups of similar frequencies [72]. The *spectral* or *frequency* components of a sound serve as input to F_1 ART layers, where a specific pattern of cells is activated with a certain sound. Bottom-up signals are thus generated and passed to a *pitch* layer (i.e., layer F_2), and so on.

We have also developed a model that extends the spatial clustering capabilities of FAST by introducing short-term memories that determine the activation of categories in a partially observable environment [160]. In other words, we add short-term memories to the FAST network to disambiguate identical sensory inputs observed in different situations. This model was motivated by the use of a FAST network in the incremental construction of the internal representation of the environment in an autonomous mobile robot [159]. It will be described in more detail in Chapter 6.

4.7 Summary

There is growing evidence that the development of the neural circuits of nervous systems depends on the inputs they receive from the environment (neural epigenesis), which increases their learning flexibility and voids the heavy burden that nativism places on genetic mechanisms. Significant studies about the brain mechanism of vision are a prime example of the role of activity and of experience in the formation of the functional properties of neurons and of the refinement of connectivity between neurons.

Similarly, in the domain of artificial neural networks a special class of learning algorithms has been introduced to deal with the artificial neural network topology problem, offering the possibility of dynamically modifying the network's structure and providing constructive incremental learning. We have developed a structure-adaptable artificial neural network architecture called FAST. Such architecture implements a learning model based on the Adaptive Resonance Theory developed by S. Grossberg as a theory of human cognitive information processing.

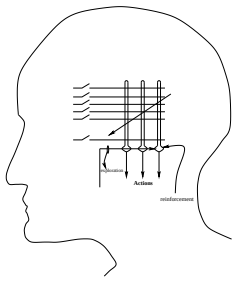
From a computational point of view, ART is an unsupervised learning model. It is particularly useful in tasks where the system is given a collection of patterns, and it is asked to construct a model to explain the observed properties of the patterns (i.e., adaptive categorization). Unsupervised learning differs from supervised learning in that it is basically a *match-based* learning process instead of an *error-based* learning [39]. In T. Poggio's words: "So far, the most ambitious unsupervised learning techniques have been used only in simple, *toy* examples but they represent the ultimate goal: learning to see, from experience, without a teacher" [162].

FAST was particularly motivated by the possibility of a digital implementation using programmable hardware devices. Such an architecture was used for image processing

and neurocontrol. In both domains, its basic function is to categorize or cluster input information by itself, according to previous experience and to a measure of similarity, much in the same way as humans recognize a given object as being part of a known class of objects. The first application served to evaluate the clustering properties of the algorithm, while the second application was our motivation to develop structure-adaptable neurocontrollers, presented in the following chapters.

Chapter 5

Reinforcement Learning: from Biology to Hardware



"The real problem is not whether machines think
but whether men do."

-B.F. Skinner

Categorization, i.e., the process by which distinct entities are treated as equivalent, is considered one of the most fundamental cognitive activities because categorization allows us to understand and make predictions about objects and events in our world. The Adaptive Resonance Theory (Chapter 4) helps to explain some sensory and cognitive processes in the brain such as perception, recognition, attention, working memory, etc. However, other types of learning, such as *reinforcement learning*, seem to govern spatial and motor skill acquisition [22].

While the ART theory claims that only resonant states can drive new learning (i.e., when the current inputs sufficiently match the system's expectations) [73], other types of brain adaptation theories suggest that "learning is driven by changes in the expectations about future salient events such as rewards and punishments" [184].

Reinforcement learning refers to a class of adaptation algorithms where the system (or agent) has to learn through *trial and error* coupled with a reward/punishment mechanism. The key concept of reinforcement learning methods is called *temporal-difference* learning, where adaptation is driven by the difference between temporally successive predictions (i.e., a change in expectations).

This computational approach offers an attractive paradigm to automating goal-directed learning and decision making. It has been successfully used for learning strategies for gaming and control. Reinforcement learning techniques are very well suited for online learning since their training examples can be taken directly from the temporal sequence of ordinary sensory inputs, unlike supervised learning, where an external

supervisor is needed.

Modern reinforcement learning techniques are the result of concepts developed for the domains of animal learning, optimal control, and, more recently, temporal-difference learning. The ideas about learning by *trial-and-error* were initially developed in research about animal learning. The concepts of *state* in a dynamic system and of *value function* are derived from the domain of optimal control (the idea of learning is notably absent in this domain of inspiration). Finally, *temporal-difference* learning originates in part from animal learning psychology, and particularly, in the notion of *secondary reinforcers* (a secondary reinforcer is a stimulus that has been associated to a primary reinforcer such as food or pain, and has come to take similar reinforcing properties) [200], even though the first implementation of such algorithms was Samuel's famous checkers-playing program (1959) [176] which apparently did not draw inspiration from such concepts, but rather from Shannon and Turing's works [189, 213].

We were particularly motivated by reinforcement learning techniques because they offer an attractive paradigm for online and continual learning without the need of an external supervisor or separated phases for training and utilization. Reinforcement learning mechanisms thus seem to be an elegant paradigm for adaptation in autonomous systems, and complement very nicely the adaptive categorization techniques explored in the last chapter.

In this chapter we will present the main concepts of reinforcement learning, the different problems that arise when dealing with such techniques and some successful applications. In Section 5.1 we start by introducing the principles of learning to predict rewards (particularly in animal learning). In Section 5.2 we introduce reinforcement learning, Section 5.3 briefly describes reinforcement learning within the context of neurocontrol and Markov decision processes, and in Section 5.4, we deal with the principles of temporal-difference learning. Section 5.5 presents the problem of learning game strategies: we used the game of Blackjack as a problem that enabled us to better understand reinforcement learning techniques and we present it to illustrate the dynamics of learning by interaction (Widrow also used Blackjack in the early 1970s to test one of the first reinforcement learning algorithms). It also allowed us to think about the use of reinforcement learning techniques for learning strategies in a more general problem (i.e., the problem of autonomous robot control). Section 5.6 presents $TD(\lambda)$ learning and model-based reinforcement learning methods, two extensions of TD-learning. Section 5.7 presents the problem of maze learning as an abstraction of the real problem of autonomous robot navigation and a comparison of several reinforcement learning techniques in such task (an original contribution of this thesis). Section 5.8 presents the problem of generalization in reinforcement learning. Section 5.9 presents a solution to the well-known inverted pendulum control problem using a novel neurocontroller architecture, and, finally, Section 5.10 presents the digital implementation of the Adaptive Heuristic Critic (AHC) model.

5.1 Learning to predict rewards

Adaptive organisms have to deal with a continually changing environment. Thus, in order to survive, they must be able to predict or anticipate future events by generalizing the consequences of their behavioral responses to similar situations experimented in the past. Predictions thus give adaptive organisms the time to prepare their behavioral reactions. Predictions are based on previous experiences after combining *search* and *memory* [200]: the adaptive organism (or agent) selects one of the many possible actions in a particular situation, and then remembers the actions that worked best and forgets the inadequate actions. The use of search and memory gives rise to what is called *trial-and-error* learning.

Trial-and-error learning was proposed by the American psychologist Edward Thorndike in the early 1900s. He attempted to develop an objective experimental method to study the mechanical problem solving ability of cats and dogs. He devised a number of wooden puzzle-boxes which could be opened by various combinations of latches, levers, strings, and treadles. A dog or a cat would be put in one of these boxes and, sooner or later, would manage to escape from it. Thorndike observed that occasionally, quite by chance, an animal performs an action which frees it from the box. When the animal finds itself in the same position again, it is more likely to perform the same action again. The reward of being freed from the box somehow strengthens an association between a stimulus (being in a certain position in the box) and an appropriate action. Reward acts to strengthen stimulus-response associations. The animal learns to solve the puzzle-box not by reflecting on possible actions and really puzzling its way out of it but by a quite mechanical development of actions originally made by chance [216].

In 1911 Thorndike proposed the so-called *law of effect*:

> *"Of several responses made to the same situation those which are accompanied or closely followed by satisfaction to the animal will, other things being equal, be more firmly connected with the situation, so that, when it recurs, they will be more likely to recur; those which are accompanied or closely followed by discomfort to the animal will, other things being equal, have their connections to the situation weakened, so that, when it recurs, they will be less likely to occur. The greater the satisfaction or discomfort, the greater the strengthening or weakening of the bond"* [208].

The law of effect is both *selective* (i.e., it involves trying alternative actions and selecting among them by comparing their consequences) and *associative* (i.e., the alternatives found by selection are associated to particular situations), while supervised learning is only associative [200].

The concept of *reinforcement* is an extension of the Thorndike's law of effect and was developed by B.F. Skinner [195]. A reinforcement is a generally positive event (e.g., the distribution of food, in the case of animal learning experiments) occurring after an action is performed, which increases the probability that such action will be performed in the future.

Neuroscience researchers have identified a neural substrate of prediction and reward in experiments with primates. The so-called *dopamine neurons* have been shown to be

"excellent feature detectors of the *goodness* of environmental events relative to learned predictions about those events" [184]. They emit a positive signal if a desired event is better than predicted, no signal if a desired event occurs as predicted, and a negative signal if the desired event is worse than predicted [184].

Moreover, Dehaene and Changeux [50] proposed that the organism itself can monitor its own success via an internal auto-evaluation loop. Such learning mechanism takes place purely by mental experiment, without any interaction with the environment (a feature essential in mathematical development and other cognitive domains).

Similarly, artificial systems can "learn to predict" by the so-called *temporal-difference* (TD) methods. Temporal-difference methods are computational models that enable a system to learn to predict rewards by trial-and-error, based on the idea that "learning is driven by changes in the expectations about future salient events such as rewards and punishments" [184]. Such changes are detected by computing temporal differences of the estimations of rewards.

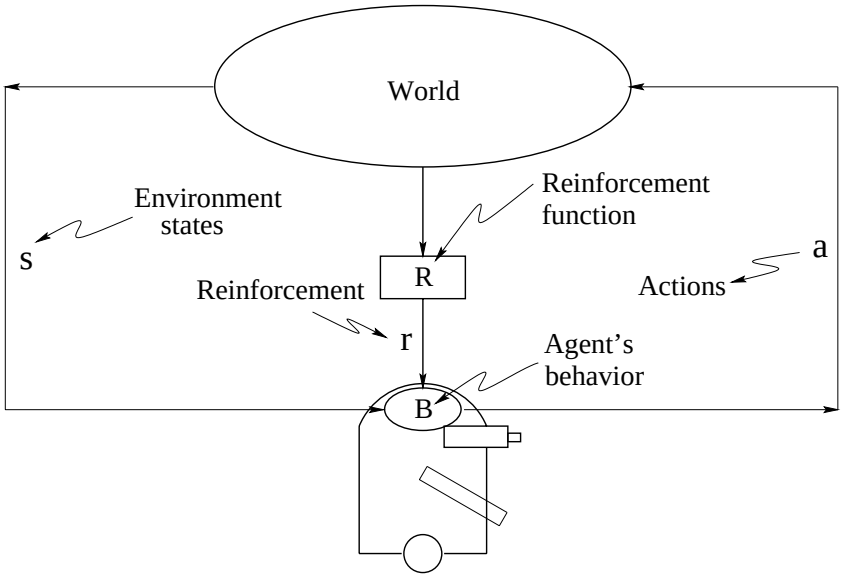


Figure 5.1: Reinforcement learning.

5.2 Reinforcement learning

Reinforcement learning refers to a class of adaptation algorithms where only a scalar evaluation (reinforcement) of the system's performance is available [23].

Reinforcement learning tasks are generally treated in discrete time steps. At each time step t, the learning system receives some representation of the environment's *state*, it *tries* an action, and one step later it is *reinforced* by receiving a scalar evaluation (i.e., a reward or a punishment) and finds itself in a new state (Figure 5.1). Reinforcement learning techniques involve the use of a *value function*: a function of *state values* or *state-action values*. A state value (denoted as $V(s)$) represents the expected cumulative

reinforcement an agent can receive starting from a given state s; an action value (denoted by $Q(s,a)$) represents the expected cumulative reinforcement an agent can receive when choosing action a from state s. In the general case, the behavior of the learning system (i.e., its *policy*) is a simple function of the value function. Therefore, adaptation is achieved by updating the value function using the scalar evaluation received while interacting with the environment. A key idea in reinforcement learning is called *temporal-difference* (TD) learning [197]. TD learning methods allow a system to learn from the raw experience without a model of its environment and to update its estimates of the value function using previously learned estimates (i.e., by *bootstrapping*).

A reinforcement learning system has to deal with the *temporal credit assignment* problem, i.e., how to punish or reward actions when they have long-reaching effects, and the *exploration-exploitation dilemma*, i.e., how to decide when to try new actions (to explore) in order to discover better action selections for the future [200].

Reinforcement learning techniques appear to be very useful in problems where the environment provides a stream of input patterns, and the target output for each input pattern are not known [75]. Applications of such learning algorithms arise in sequential-decision processes, such as the development of optimal strategies for gaming and control.

5.3 Neurocontrol and Markov decision processes

A major problem in nonlinear control is the tuning and adaptation of the controller. For this purpose a model of the process is usually developed. Then, following an approximation of the inverse relation between the desired outputs and the control actions, the controller is adjusted. However, for many real-world problems there is no available quantitative data regarding input-output relations, rendering analytical modeling very difficult [27]. Moreover, errors in the model can lead to poor performance of the controller. Artificial neural networks, and particularly reinforcement-learning-based neural networks have been proposed as an alternative for implementing adaptive controllers which are capable of "learning" (i.e., *neurocontrollers*) [182].

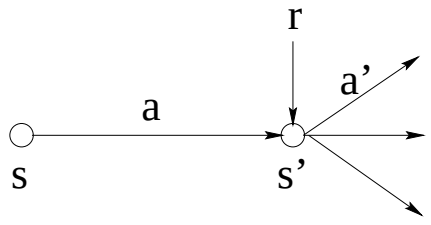


Figure 5.2: State transition in a Markov decision process (MDP).

Neurocontrol problems can be modeled as Markov decision processes (MDPs). A Markov Decision Process is a discrete stochastic version of the so-called *optimal control* problem, introduced by Bellman in the late 50's [24]. The optimal control problem concerns the design of a controller that minimizes a measure of the dynamical behavior of the system (i.e., by solving the so-called Bellman equation). The class of methods for solving optimal control problems is known as *dynamic programming* [200].

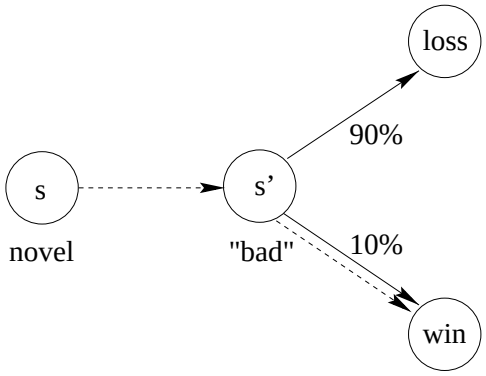


Figure 5.3: Board game example. The problem consists of determining if a novel game board position s is "good" or "bad" given that the following position (s') is known to be "bad" from previous experience (since 90% of the time it leads to a loss and 10% of time to a win) and that the first time through, the game evolves as indicated by the dashed arrows.

A Markov Decision Process is composed of a set of possible states S, a set of actions A, a reinforcement function $R : S \times A \rightarrow R$, and a state transition function $P : S \times A \rightarrow \Pi(S)$ which determines probabilistically the next state of the environment as a function of the current state and the selected action. $\Pi(S)$ is a function that maps states to probabilities. For example, $P(s, a, s')$ represents the probability of making a transition from state s to state s' taking action a (Figure 5.2).

When the state transitions are independent of any other state transitions or previous selected actions, the model is said to be *Markovian* [104]. Reinforcement learning techniques can be successfully applied to solve Markov decision processes, as will be shown in the next sections.

5.4 Temporal-difference (TD) learning

The so-called *temporal-difference* methods are a class of incremental learning algorithms specialized for prediction, that is, "for using past experience with an incompletely known system to predict its future behavior" [197].

The first uses of temporal-difference methods include Samuel's checkers player in the late 1950s [176], Holland's bucket brigade algorithm in the middle 1970s [87], and Barto, Sutton, and Anderson's adaptive heuristic critic (AHC) learning algorithm in the early 1980's [23]. However, such algorithms were poorly understood until Sutton's paper "Learning to Predict by the Methods of Temporal Differences" in the late 1980s [197].

Learning in TD methods is driven by the difference between temporally successive predictions (i.e., the system learns when there is a change in prediction over time). A key advantage of TD methods is that their training examples can be taken directly from the temporal sequence of ordinary sensory inputs. They have been successfully used to learn strategies for gaming and control. To illustrate the operation of TD learning, Sutton [197] presented a very simple game-playing task, shown in Figure 5.3.

Supposing that a game position s' is known to be "bad" by previous experience, and that, the first time a game position s occurs, the game evolves as shown in Figure 5.3, the

problem would be to know if such novel game position should be evaluated as a "good" one or a "bad" one as a result of such experience. A supervised learning method, for example, would form a pair (input,desired-output) from the novel state s and the *win* state that followed it, and would class the novel game position as a "good" one, whereas a temporal difference learning method would form a pair from the novel state s and the following state s', which is known to be a "bad" one (thus classifying it correctly).

TD methods exploit the information provided by the temporal sequence of states (i.e., the game board positions), which is ignored by most supervised learning algorithms. TD methods are particularly useful in *multiple-step* prediction problems, that is, problems where the information about the correctness of the predictions is not revealed until more than one step after the prediction is made [197].

The problem of distributing the credit of success among the many decisions taken by the system which are involved in producing it is known as the *credit assignment problem* (after Minsky in the early 1960s). In the following sections we present some widely used TD methods and some applications we have explored in this thesis in order to understand the dynamics of such algorithms and their usefulness for autonomous learning in real world problems.

5.4.1 Q-learning and SARSA learning

Q-learning [222] is a temporal-difference (TD) method that attempts to solve the temporal credit assignment problem. In Q-learning, the learned action-value function Q directly approximates the optimal action-value function, denoted by $Q^*(s,a)$. This can be achieved by modifying the Q values such that the *temporal-difference error* is minimized :

$$Q(s,a) = Q(s,a) + \alpha TD_{err} \ , \tag{5.1}$$

$$where \quad TD_{err} = r + \gamma max_{a'} Q(s',a') - Q(s,a) \ , \tag{5.2}$$

α is a constant parameter between 0 and 1, known as the *step-size*.

The value of $Q(s,a)$ is thus updated so that its value tends to the sum of the current reward r and the maximum future reward $max_{a'}Q(s',a')$ discounted by $0 < \gamma < 1$, that is, $Q(s,a) \rightarrow r + \gamma max_{a'}Q(s',a')$. This expression is an estimate of the total reward the agent can receive by taking action a from state s. The development of Q-learning and the mathematical proof of its convergence with probability 1 (under certain assumptions) has been one of the most important breakthroughs in reinforcement learning [200].

The SARSA algorithm [175] is also a temporal difference (TD) method. At every time step, SARSA updates the estimations of the action-value functions $Q(s,a)$ using the quintuple (s,a,r,s',a'), which gives rise to the name of the algorithm. Unlike Q-learning, SARSA does not estimate future reward as a function of the discounted maximum possible reward of taking action a' from next state s' (i.e., $\gamma max_{a'}Q(s',a')$) to estimate the total future reward of taking action a from state s, but simply as a function of $\gamma Q(s',a')$. Figure 5.4 presents the SARSA algorithm in procedural form.

1. Initialize the $Q(s, a)$ value functions arbitrarily,
2. Select an initial state s,
3. Select an action according to a certain policy
4. Take action a; observe the reward (r), find the next state s', and select the next action a',
5. Update the estimate value $Q(s, a)$: $Q(s, a) = Q(s, a) + \alpha TD_{err}$, where $TD_{err} = r + \gamma Q(s', a') - Q(s, a)$ is the temporal-difference error, α is the step size, and γ is a discount reward factor,
6. Let $s = s'$,
7. Go to step 3 until the state s is a terminal state,
8. Repeat steps 2 to 7 for a pre-defined number of reruns.

Figure 5.4: SARSA algorithm for estimating action-value functions.

5.4.2 Adaptive heuristic critic (AHC) learning

The AHC algorithm consists of two elements (Figure 5.5): (1) an Associative Search Element (ASE) or simply the *actor*, whose output is the action, and (2) an Adaptive Critic Element (ACE), or simply the *critic*, which builds predictions of the reinforcement signal (i.e., the expected performance) [23].

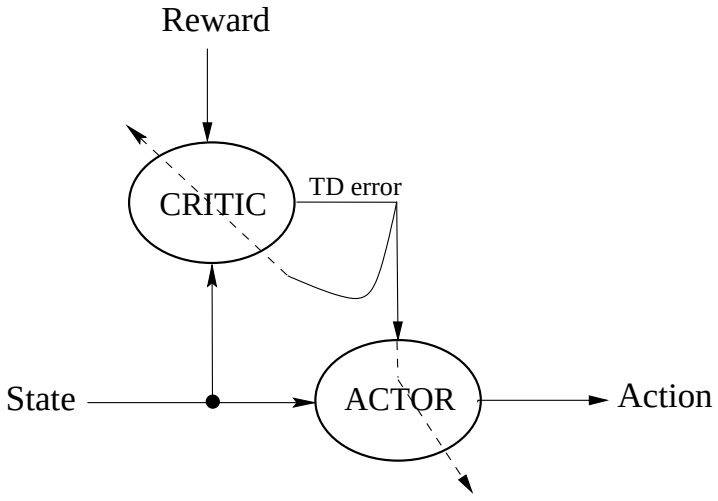


Figure 5.5: Adaptive heuristic critic (AHC) model.

Essentially, the critic implements a state-value function that evaluates the new state after taking an action, to determine if the system acted better or worse than expected. Such evaluation is based on the TD error, which is computed as follows:

$$TD_{err} = r + \gamma[V(s') - V(s)] \ , \tag{5.3}$$

where V is the current value function implemented by the critic, r is the reinforcement signal, and γ is the discount factor. When the TD error is positive, it suggests that the probability of taking action a should be increased for the future and *vice versa* [200]. The TD error thus drives adaptation in the critic itself as well as in the actor. The critic is updated as follows:

$$V(s) = V(s) + \alpha TD_{err} \ , \tag{5.4}$$

where α is a constant parameter (usually referred to as the step-size).

The actor implements the policy (i.e., the behavior of the system) by means of modifiable parameters that indicate the preference $p(s, a)$ of taking the action a given a state s. Such preferences are updated proportionally to the TD error:

$$p(s, a) = p(s, a) + \beta TD_{err} \ , \tag{5.5}$$

where β is a constant parameter (also a step-size). An action a is selected depending on the preferences $p(s, a)$ using an ϵ-greedy or a *softmax* mechanism (Subsection 5.4.3).

Actor-critic methods were developed in the early 1980s but have not been extensively used. On the contrary, Q-learning and SARSA have received a lot of attention, even though they cannot learn a policy independently of the value function as is the case with actor-critic methods [200]. Despite the advantages of actor-critic models, Q-learning and SARSA are less computationally intensive, since the policy is obtained directly from the value function and need not be separately computed.

5.4.3 Exploration vs. exploitation

One of the problems that arises in the domain of autonomous learning systems is how to decide when to try new actions (i.e., when to explore) in order to discover better action selections for the future. Such problem is often known as the *exploration-exploitation dilemma*. Reinforcement learning methods predict rewards by exploiting value functions that map states or state-action pairs to rewards (Section 5.2). When the system chooses the action a that maps to the highest prediction, we say that the system is *exploiting*, whereas when it selects another action in order to gain new information, we say that it is *exploring*. One of the first researchers to investigate it in the domain of adaptive systems was Holland [86]. He observed that a system may lose performance by searching new information (exploring) while it may never ameliorate its performance if it just exploits the best of what it has previously learned (exploiting).

The simplest explore/exploit strategy consists on taking the action with highest Q-value with a fixed probability $1 - \epsilon$, where ϵ is a constant ($0 < \epsilon < 1$), and a random action otherwise. This method is called the *ϵ-greedy* action selection mechanism. It is very easy to implement and enables the system to learn continually. However, in *stationary* problems (i.e., a problem in which its parameters do not change in time) the continual exploration may lead to suboptimal results, even if the optimal solution has already been learned.

A second explore/exploit strategy consists on taking the action with highest Q-value with a probability that is a function of time. Typically, such probability tends to one asymptotically over time.

A third explore/exploit strategy is called the *softmax* action selection. The probability $p(a_i)$ of choosing action a_i is given by the following expression:

$$p(a_i) = \frac{e^{Q_t(a_i)/\tau}}{\sum_{a_j=1}^{n} e^{Q_t(a_j)/\tau}} , \tag{5.6}$$

where τ is a positive parameter called the *temperature*. At high temperatures all actions are nearly equally likely, while at low temperatures the probability of taking a given action is more dependent on the value estimates. In the limit $\tau \to 0$, the softmax selection becomes the $\epsilon - greedy$ action selection. One problem of the softmax action selection mechanism is how to determine the value of τ, which depends on the task [200].

Further explore/exploit strategies based on a prediction error or its rate of change are presented in [235]. Moreover, Thrun [209, 210] distinguishes between *undirected* and *directed* exploration. The first class of strategies is characterized by the use of randomness for exploration, whereas the second is characterized by the use of exploration-specific knowledge for guiding exploration. The idea of the directed exploration is to optimize the knowledge gain of the environment by taking the actions that are most informative for the learning task at hand, since it is impossible to determine the result of actions in an unknown or partially-unknown environment. In particular, a simple strategy to gain knowledge of the environment is to try the actions/states that have been less frequently or less recently selected. Directed exploration has been shown to be more efficient in learning time [209].

5.5 Learning game strategies

In the early 1970s, Bernard Widrow used the game of Blackjack to test one of the first reinforcement learning algorithms, called *selective bootstrap adaptation*. He demonstrated how an ADAptive LINear Element *learned with a critic* to play Blackjack without knowing the game or the objective of play [228]. Reinforcement learning techniques have since been widely used for learning many games: Backgammon [204], Checkers [176], Go [183], Chess [114], Tic-tac-toe [29], etc.

We have used Q-learning [222] and SARSA [175] to learn to play Blackjack in order to better understand reinforcement learning techniques, and to explore the idea of learning strategies not only for gaming but also for control. Our experiences with reinforcement learning and Blackjack [158] are similar to those realized in [150]. However, we used a slightly different state encoding, which enabled us to better analyze the learned strategies. Since we did not find important differences between Q-learning and SARSA solutions in the Blackjack task, we will only refer to the SARSA learned strategies.

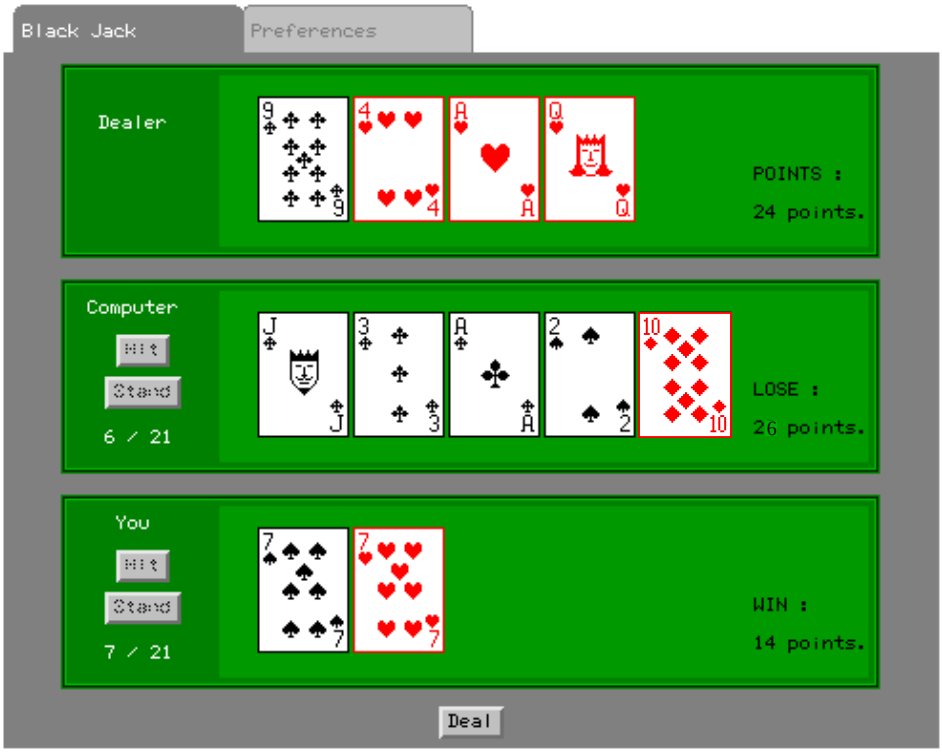


Figure 5.6: Blackjack learning.

5.5.1 The game of Blackjack

Blackjack or Twenty-One is a card game where the player attempts to beat the dealer by obtaining a sum of card values higher than the dealer's and equal to or less than 21. Each card has the same value as its index except for the ace (which can be counted as 1 or 11, as the player's choice) and the picture cards (which are counted as 10). The ace can be valued as either 1 or 11, at the option of the player. At the beginning of the game, each player is dealt two cards, one face up and one face down. After looking at his first two cards, the player chooses to draw (hit) or to stop drawing cards (stand). The player may take as many hits as he wants as long as he doesn't "bust", i.e., as long as the sum of card values in his hand does not exceed 21. Once all the players have finished their hands, the dealer shows his or her face-down card and draws cards until he/she has a total of 17 or above (the standard strategy in professional Blackjack).

In our experiments, the game of Blackjack was simplified by removing interesting aspects, such as betting, and special features, such as "splitting pairs", "doubling down", "insurance", etc. The game has one dealer, and one or two players. The deck of 52 cards is shuffled before each hand. An ace is automatically valued as 11, unless it would cause a bust, in which case it is valued as 1. The state s of our system corresponds to the sum of card values in the player's hand (i.e., the score), if the player does not have an ace, $s = \{4, 5, 6, ..., 20\}$. If the player has an ace, there is a set of 9 extra states $s = \{23, 24, ..., 31\}$ determined by the sum of card values being held plus 11 (note, that states 23 to 31 correspond to a sum of the card values in hand between 2 and 10, when

Q/S	4	5	6	7	8	9	10	11	
$Q(s,0)$	-0.198	-0.281	-0.267	-0.293	-0.221	-0.098	-0.01	+0.005	
$Q(s,1)$	-0.203	-0.289	-0.301	-0.281	+0.206	-0.124	-0.042	-0.532	
Q/S	12	13	14	15	16	17	18	19	20
$Q(s,0)$	-0.327	-0.324	-0.388	-0.589	-0.449	-0.464	-0.190	-0.342	-0.090
$Q(s,1)$	-0.042	-0.561	-0.532	-0.552	-0.440	-0.410	-0.573	-0.114	+0.249

Table 5.1: Learned state-action values $Q(s,a)$ after approximately 500 runs of 100 games. The set of actions are '0' (to hit) and '1' (to stand). We show the action-values up to state $s = 20$ for clarity.

considering the value of an ace as 1). Two terminal states, $s = 21$ and $s = -1$, were also introduced to indicate a "perfect" score or a bust. At each stage of the game, the player must choose whether to hit or to stand. This selection follows an ϵ-greedy policy.

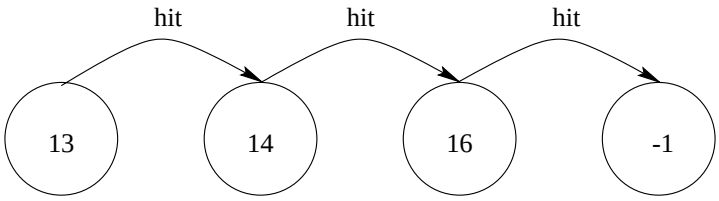


Figure 5.7: State transitions of a Blackjack game example. The system starts with a figure and a three in hand (state 13), it chooses to hit, receives an ace, and changes to state 14. In state 14, it chooses to hit, and changes to state 16 after receiving a two as new card. Finally, it chooses 'to hit' from state 16 and falls into the terminal state -1.

5.5.2 Example of learning

The learning system consisted of a look-up table implementation of the SARSA algorithm, using the environment's state encoding previously described. In our first experiments, the reinforcement signal was defined as follows:

$$r = \begin{cases} -1 & if\ bust, \\ +1 & if\ win, \\ 0 & otherwise \end{cases}$$

In Table 5.1 we present the current state-action values of a system that has been playing the game of Blackjack for nearly 50,000 times. The discount factor was set to 0.9 and the step-size to 0.01. If we consider the hand shown in Figures 5.6 and 5.7, where our system plays first, then a human player, and finally the dealer, we notice that our system has followed the sequence of states: $s = 13$, $s = 14$, $s = 16$, and finally $s = -1$ (a bust). Learning takes place by using the SARSA-learning algorithm as follows:

1. The initial state of the system corresponds to the sum of card values in hand: $s = 13$.

2. The system chooses an action according to a certain policy, for example, choosing the action with the highest Q value. Since $Q(13, hit)$ equals -0.324 and $Q(13, stand)$ equals -0.561, the system takes action 'to hit'.

3. The player receives an ace as new card and thus modifies its state to $s = 14$.

4. To update the state-action value $Q(13, hit)$ we follow the standard SARSA update:

   $Q(13, hit) = Q(13, hit) + \alpha(r + \gamma Q(s', a') - Q(13, hit))$, which requires to select a next action a'. Assuming that the next action is chosen to be 'to hit', $Q(13, hit)$ is updated using the estimate of the reward of the next state-action $Q(14, hit)$ as follows:

   $Q(13, hit) = -0.324 + 0.1[0 + (0.9 \times -0.388) - (-0.324)] = -0.326$

5. The system thus takes the action 'to hit' from state $s = 14$ and receives a two as new card. The systems modifies its state to $s = 16$.

6. Similarly to step 4, the system updates $Q(14, hit)$ as a function of $Q(16, a')$, where a' is the action chosen from state $s = 16$, which happens to be 'to hit', given that $Q(16, hit)$ is larger than $Q(16, stand)$, as a result of the learning during the nearly 50,000 games.

7. The systems takes action 'to hit' from state $s = 16$ and receives a ten as new card, resulting in a bust. The system falls in the terminal state $s = -1$, and updates $Q(16, hit)$ as follows:

   $Q(16, hit) = -0.449 + 0.1(-1) = -0.459$ (r=-1)

The present hand served to learn the taking of the action 'to stand' from state $s = 16$, given that the new state-action value of $Q(16, hit)$ is higher than the value $Q(16, stand)$. This is generally the case, given that the action 'to hit' leads to a bust with higher probability. If the system would stand in step 7 of this hand, it would obtain a score of 16, which will allow it to win the game against the dealer.

5.5.3 Experimental results

The basic rules of the Blackjack are quite simple. However, it is not evident to determine an optimal playing strategy. In the first experiments (one experiment consisted of 1000 runs of 100 games) the simulated player used a set of fixed strategies. The results of such experiments (Table 5.2) enabled us to rank the learned strategies.

Table 5.2 presents the overall percentage of won games and the maximum and minimum performance during the 1000 runs for a player using the dealer's strategy, a conservative strategy (hold) where the player always stands, and a random strategy (always against a dealer with the fixed strategy described above).

Table 5.4 presents the performance results of the algorithm with different values of ϵ: 0.1, 0.01, and a decaying exploration probability: $\epsilon(r) = 0.99^r * \epsilon(0)$, where $\epsilon(0) = 0.1$, and r is the current run ($0 < r < 1000$). It can be seen that a high exploration probability ($\epsilon = 0.1$) can find temporary solutions with very good results. However, this may lead

Strategy	Avg(%)	Max(%)	Min(%)
dealer's	40.7	57	25
hold	38.3	51	24
random	31.5	46	18

Table 5.2: Performance of fixed strategies against the dealer's strategy.

Q/S	4	5	6	7	8	9	10	11	
$Q(s,0)$	+0.063	+0.850	+0.860	+0.870	+0.880	+0.884	+0.899	+0.928	
$Q(s,1)$	+0.985	+0.068	+0.059	+0.068	+0.149	+0.149	+0.182	+0.317	
Q/S	12	13	14	15	16	17	18	19	20
$Q(s,0)$	+0.245	+0.078	-0.016	-0.112	-0.191	-0.208	-0.240	-0.223	-0.355
$Q(s,1)$	+1.000	+1.000	+1.000	+1.000	+1.000	+1.000	+1.000	+1.000	+1.000
Q/S	23	24	25	26	27	28	29	30	31
$Q(s,0)$	+0.831	+0.855	+0.860	+0.872	+0.0870	+0.881	+0.899	+0.901	+0.903
$Q(s,1)$	+0.059	+0.059	+0.030	+0.030	+0.077	+0.086	+0.077	+0.068	+0.077

Table 5.3: Learned state-action values $Q(s,a)$ after 1000 runs of 100 games. The set of actions are '0' (to hit) and '1' (to stand).

to reduced overall performance, sometimes even lower than the dealer's fixed strategy. The overall performance after 1000 runs of 100 games of the SARSA algorithm with 0.01-greedy selection action mechanism (Table 5.4) surpasses the average rate of success of 40.6% reported in [150], and the performance of fixed strategies presented in Table 5.2. Note that our approach uses punishment *when bust* instead of *when loss*, which gives rise to a more conservative strategy but higher rates of success.

Performance of learned strategies

Using a discount factor of $\gamma = 0.9$, a step size $\alpha = 0.01$, and an ϵ-greedy action selection $\epsilon = 0.01$, the SARSA learning algorithm discovered the following strategy:

$$action = \begin{cases} \textbf{hit} & if\ (score < 11)\ or\ (an\ ACE\ is\ held) \\ \textbf{stand} & otherwise \end{cases}$$

Such strategy can be interpreted from the learned action-values after the 1000 runs of 100 games, given in Table 5.3 (as a matter of fact, those Q values indicate that in state 4, the learner should not hit, an anomaly which might be explained by the relative rarity of such state due to the low probability of receiving two consecutive two-valued cards at the beginning).

The learned strategy is very conservative, i.e., not to hit if the score is larger than 11. However, it is interesting to note how the algorithm determines such threshold without explicitly programming the rules of the game (just by experience and the reinforcement signal at the end of each hand). In fact, the SARSA player learns to hit only if it is sure

ϵ	Avg(%)	Max(%)	Min(%)
0.1	41.1	56	26
0.01	42.4	56	26
$0.1 * 0.99^r$	40.9	53	26

Table 5.4: Performance of learned strategies against the dealer's strategy.

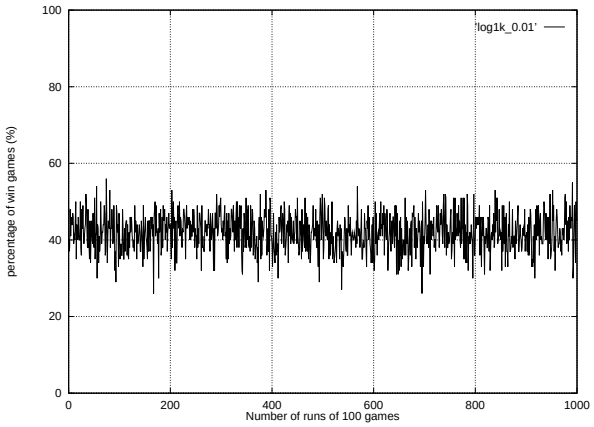


Figure 5.8: Percentage of won games during 1000 runs of 100 games.

that it will not bust, and it discovered by experience the double card value of the ace. Figure 5.8 shows the percentage of won games during the 1000 runs of a SARSA learner with 0.01-greedy action selection mechanism.

5.6 $TD(\lambda)$ learning and model-based methods

Temporal-difference (TD) learning methods like Q-learning attempt to solve the temporal credit assignment problem. However, these methods suffer from the curse of exponential growth in computation time as a function of the number of states [25]. As a consequence, function approximation techniques have been used to provide generalization across states and actions [200] (Section 5.8). Nevertheless, several methods, such as $TD(\lambda)$ and $Dyna$, have been proposed to deal with this problem and allow tabular implementations. In particular, look-up tables are easier to implement than function approximators used for generalization (Section 5.8).

From a computational point of view, both TD(λ) and Dyna help the system learn more efficiently by updating a set of state/action values rather than a single value in every time step.

5.6.1 $SARSA(\lambda)$: reinforcement learning with eligibility traces

The TD(λ) algorithm, proposed by Sutton in [197], uses *eligibility traces* that serve as a temporary record of the occurrence of an event, such as visiting a state or taking an action (Figure 5.9), helping the system to more efficiently distribute rewards or penalties

to previously visited states or taken actions (i.e., to handle delayed rewards/penalties). In general, the older the actions or visited states, the smaller their effect on the resulting behavior.

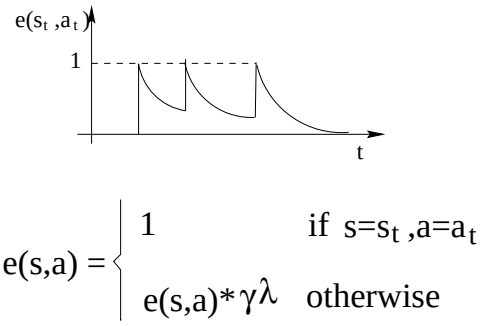


Figure 5.9: Example of eligibility traces. Whenever an action a_t is taken from state s_t, its corresponding trace $e(s_t, a_t)$ is set to 1. Otherwise, the trace decays exponentially. γ and λ are constants in the range (0 1].

Eligibility traces can be easily integrated to SARSA learning methods, giving rise to SARSA(λ). In this method, all the Q action values are modified after every interaction with the environment as follows:

$$Q(s,a) = Q(s,a) + \alpha[r + \gamma Q(s',a') - Q(s,a)]e(s,a) \ , \tag{5.7}$$

where s is the state, a is a possible action, s' and a' are the corresponding possible next state and action, r is the reward, α is a step-size parameter, and $e(s,a)$ is the eligibility trace of action a taken from state s. An eligibility trace is set to 1.0 when action a is taken from state s and updated as $e(s,a) = \gamma^{\lambda} e(s,a)$ after every interaction with the environment. γ is the discount factor and λ is a constant value in the range $(0, 1]$.

5.6.2 Dyna-SARSA: a model-based reinforcement learning

The *Dyna* model [198] integrates TD and *planning*. It uses a model of the environment to improve the interaction with the actual environment. Dyna (Figure 5.10) alternately operates on the environment and on the learned model of the environment: an action value is updated after every interaction with the world and after a certain number of interactions with the dynamically created model of the environment (these latter are called *planning updates*). The following expression is the Dyna-SARSA action value update:

$$Q(s,a) = Q(s,a) + \alpha[r + \gamma Q(s',a') - Q(s,a)] \ , \tag{5.8}$$

where s is the state, a is a possible action, s' and a' are the corresponding possible next state and action, r is the reward, and α is the step-size parameter.

In Markovian tasks, Dyna memorizes the next state s' and the corresponding immediate reinforcement signal r received after taking action a from state s. In other words, a look-up table is addressed by s and a to retrieve s' and r such that $Q(s,a)$ could be

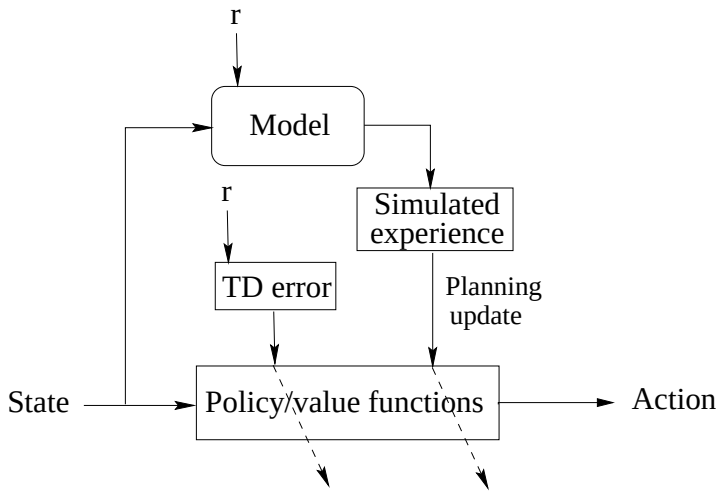


Figure 5.10: Dyna architecture.

updated as a "mental rehearsal" process. In non-Markovian tasks or partially observable problems, the model of the world is built by means of a look-up table: for each pair (state,action), the model stores all observed next states along with their frequency of occurrence and an average of the received reward. When using the model, the next state s' is computed probabilistically according to the observed frequencies [113].

These algorithms are particularly useful in problems where computation is not a major issue but real-world experience is very costly [104]. A number of new algorithms exploiting this idea and optimizing the use of the internal model of the world have been proposed [11, 142, 151], but we have not analyzed if they are suitable for a digital implementation.

5.7 Maze navigation tasks

Maze tasks have been used to study trial-and-error learning since the early work of Shannon in the 1950s [190] as an abstraction of animal learning experiments (Section 5.1). Maze learning is presented here as an abstraction of the real navigation problem in mobile autonomous robots, discussed in Chapter 6: we have an a-priori partition of the sensor space of the robot, a perfect vision, and a discrete world. An autonomous robot is faced to the problem of finding a goal (G) starting from a valid position (S) in a labyrinth-like environment (Figure 5.11). The robot can detect an obstacle to the north, south, left and right, and may move by one position in these same directions. The problem is to find the correct actions to take in a given state of the environment such that the robot avoids obstacles and find the shortest path to the goal.

Solving the maze of Figure 5.11 is a Markovian task if the robot knows its exact position within the maze. However, if the robot does not have access to its global position in the maze and has to make decisions based only on its sensor readings, it has *partial observability* of its environment, and the maze is non-Markovian. In real mobile robotics, the agent has only partial information about the current state of the environment, that is, it does not know the state of the whole world from the state of the sensory inputs alone

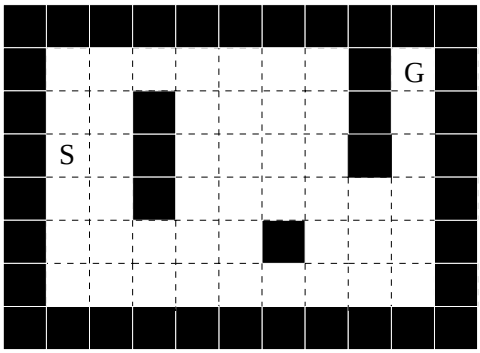


Figure 5.11: Maze navigation task. (S=start, G=goal)

(i.e., the observations). The agent is said to suffer from the problem of *hidden-state* [116] or *perceptual aliasing* [227], and the environment is said to be partially observable. A partially observable (non-Markovian) maze navigation task is a better abstraction of the real navigation task than the Markovian mazes. In the following section we present an empirical comparison between SARSA(λ) and Dyna-SARSA in both Markovian and non-Markovian tasks.

5.7.1 Markovian maze learning

To test SARSA(λ) and Dyna-SARSA, we used the maze of Figure 5.11 and a robot having access to its global position in the maze. In each trial, the learning system has to reach the goal in less than 10,000 steps in order to succeed. Otherwise, it fails, returns to the start position, and restarts a new trial. The system receives a reward of +1 if the system reaches the goal, otherwise it receives no reward. The Q-values of the system were initialized to +1.0, which is an optimistic Q-value and stimulates exploration, since, no matter which action is initially selected, the corresponding reward is less than expected, thus motivating the selection of another action [200]. In our tests, we first let the learning system interact with the environment during 1,000 trials. We then measured the average performance of the system (i.e., the number of steps needed to reach the goal) over 100 runs, each run consisting of 100 trials.

In Table 5.5 we present the resulting mean of steps needed to reach the goal in the 9x6 maze of Figure 5.11. SARSA(0.9) (i.e., $\lambda = 0.9$) and Dyna-SARSA with K=10 (K is the number of planning updates performed after each trial) presented similar performances. In Table 5.5, we also present the results of both algorithms in a 18x12 maze. In this case, Dyna-SARSA resulted in a better mean of steps. Finally, a similar 36x24 maze was used to test both algorithms. Dyna-SARSA again obtained better performance. These results indicate that Dyna-SARSA performs better than SARSA(λ) as the size of the state space increases. Furthermore, the computational load of SARSA with eligibility traces is considerably higher than that of Dyna: in a Sun Ultra-Sparc 1 workstation, Dyna-SARSA with K=10 required 7 seconds to finish the 100 runs in the largest maze, while SARSA(λ) required 1159 seconds.

Maze	Learning algorithm	Avg(steps)	Std-dev	Min(steps)	Shortest path(steps)
9x6 maze	Dyna-SARSA (K=10)	14.165	0.053	14.04	14
	SARSA(0.9)	14.22	0.076	14.07	14
18x12 maze	Dyna-SARSA (K=10)	29.39	0.105	29.2	29
	SARSA(0.9)	31.78	1.05	31.2	29
36x24 maze	Dyna-SARSA (K=10)	60.45	3.88	59.49	59
	SARSA(0.9)	66.93	26.66	61.7	59

Table 5.5: Performance of Dyna-SARSA and SARSA(λ) in a Markov maze. K is the performed number of planning updates after each trial.

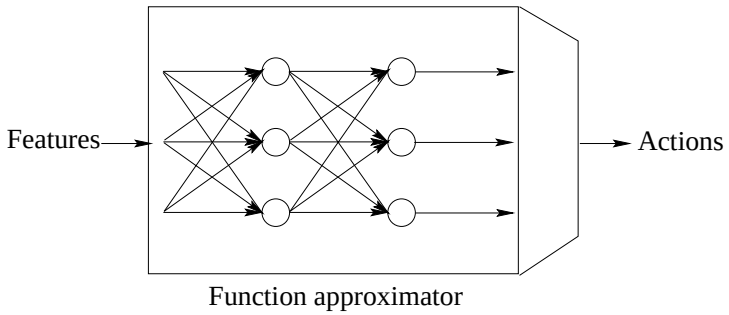


Figure 5.12: Approximation of the value function.

5.7.2 Partially observable maze navigation tasks

To test Dyna-SARSA and SARSA(λ) into a partially observable maze, we used Sutton's Markov maze [198] (Figure 5.11) and made it a partially-observable Markov decision process (POMDP) by not allowing the agent to know its exact position in the maze (as proposed by Littman [119]). The agent observes its 8 neighboring grid squares yielding 256 possible observations, even though only 30 are applicable to this maze. As with the Markov maze, we tested SARSA(0.9) and Dyna-SARSA with K=10 on 9x6, 18x12, and 36x24 mazes.

In these tests, Dyna-SARSA with K=10 learned a path of 14.1 steps in average after 24,600 trials, while the shortest path learned by SARSA(0.9) consisted of 16.06 steps in average (after 29,600 trials). However, the performance of both algorithms presented significant differences when several runs were considered: Dyna-SARSA learned a path of 251.27 steps in average, while SARSA(0.9) learned a path of 23.86 steps in average, after testing both algorithms during 300 runs of 100 trials (i.e., 30,000 trials). This means the Dyna framework was more sensitive to partial observability and most of the time did not learn a good policy. On the other hand, SARSA(λ) did not achieve the optimal policy (the optimal memoryless policy needs 14 steps to reach the goal), but did learn good policies even if the environment was not Markovian (See Loch et al. [121] for related work).

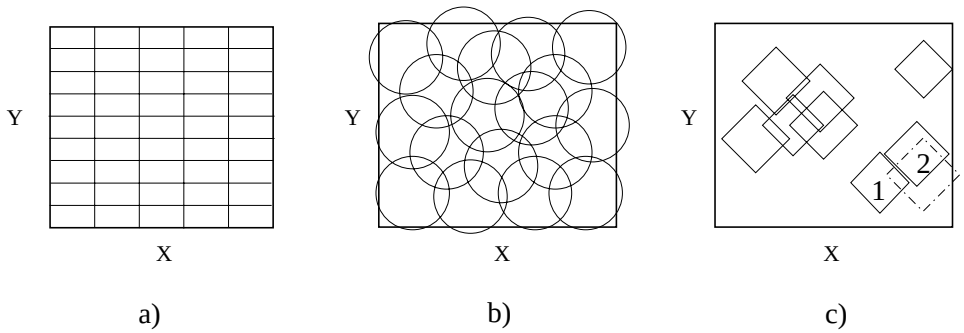


Figure 5.13: a) Tile coding, b) and c) coarse coding of a continuous 2D input space. States 1 and 2 in c) have one feature in common (the dashed tile), so there will be little generalization between them.

5.8 Approximation of value functions

The previous sections have introduced the use of value functions to map states to actions in a learning system. In particular, it was assumed that a look-up table could represent such a value function. However, this is only true for tasks with small numbers of states and actions, since the time and the amount of data needed to correctly fill such tables grow exponentially [200]. Learning in large spaces is thus realized through *generalization* techniques.

The kind of generalization required in reinforcement learning is often called *function-approximation* [200], and allows compact storage of learned information and transfer of knowledge between "similar" states and/or actions [104]. Function-approximation techniques include global function approximators like MLP (Multi-layer Perceptrons) networks, local function approximators like RBF (Radial Basis Functions) networks and coarse-coding approaches like the CMAC (Cerebellar Model Articulation Controller) model, unsupervised clustering techniques, etc.

The standard approach has been to approximate the value function and to determine a policy from it [200] (Figure 5.12). However, some researchers have also developed different approaches that will not be considered in this thesis [201, 232]. To approximate the value function, a set of features (i.e., a representation of the state) is used as input to the approximator. An estimation of the state is thus computed by the system, and the outputs of the approximator correspond to state-value or state-action pair estimations.

Generalization techniques are also necessary when the state set of a task is continuous. For example, the state of a mobile robot is determined by the readings of its sensors, which correspond to analog values. A first method for generalizing is to obtain a discrete representation of such continuous state variables. This method is known as *tile coding* and will be briefly described in the following subsection.

5.8.1 Tile and coarse coding

Tile coding (Figure 5.13a) consists of determining exhaustive partitions of the input space, i.e., of the features that are used to estimate the state of the system. Each tile is the receptive field of a binary feature (i.e., the feature is present or absent) and exactly

one feature is present in each tiling. The simplest way to tile the input space is to define a uniform grid as in Figure 5.13a.

Coarse coding (Figures 5.13b, 5.13c) is a technique that represents a state by overlapping features or receptive filters. Similarly, *state aggregation* is a technique that groups states together and associates a common table entry (a *value estimate*) to each group. When the system is in a given state in a group, the group's entry is used to determine the state's value, and when the state is updated, the group's entry is updated [200].

The cerebellar model of articulation controller (CMAC) [1] is an example of coarse coding with multiple, overlapping grid tilings. CMAC operates as a complex look-up table that can be used to approximate the value function in reinforcement learning problems [111]. Every tile is defined by quantizing functions that operate on each input, with each quantizing function corresponding to a component of the desired output value (Figure 5.14). The sum of the weights indexed by these components makes up the output value [206]. Following the example of Figure 5.14, input point A activates the quantizing functions a, b, and c with values 1, 2, and 2 along the x input variable (feature) and the quantizing functions d, e, and f with values 1, 1, and 1 along the y input variable. The CMAC output value for input A is:

$$Z_A = w_{a1d1} + w_{b2e1} + w_{c2f1} \ , \tag{5.9}$$

where w_{a1d1} is the indexed weight by the activated components (a1 and d1) in the quantizing functions a and d of the first tiling, w_{b2e1} is the indexed weight by the activated components (b2 and e1) in the quantizing functions n and e of the second tiling, and w_{c2f1} is the indexed weight by the activated components (c2 and f1) in the quantizing function of the third tiling.

Similarly, the output value for input B is $Z_B = w_{a2d1} + w_{b2e1} + w_{c2f2}$, which shares the indexed weight w_{b2e1} with Z_A, thereby illustrating how CMAC generalizes. The weights can be updated using TD learning.

5.8.2 Multi-Layer Perceptrons (MLPs)

The Multi-Layer Perceptron is a particular kind of feed-forward artificial neural architecture (Section 2.4). Given a sufficient number of hidden neurons (i.e., neurons in the hidden layer) it has been shown that an MLP is capable of approximating any nonlinear function to arbitrary accuracy [89]. A *gradient-descent* method, typically the *backpropagation* algorithm is used to adapt the interconnection weights in the MLP, so that the function being implemented more closely approximates a desired target function.

The use of gradient-descent methods with reinforcement learning techniques dates back at least to the works of Anderson and Sutton [9, 197]. An example of the application of gradient-descent and reinforcement learning techniques is the *TD-Gammon* algorithm developed by Tesauro [204]. He used a three-layered MLP network with 80 hidden neurons and TD learning to implement a Backgammon game-learning program (a problem with about 10^{20} states) that trained itself. The program played against itself and learned from the outcome. In 1992, TD-Gammon achieved near-parity against Backgammon human grand-master Bill Robertie after a 40-game test session [205]. Perhaps the most widely-known success story of reinforcement learning.

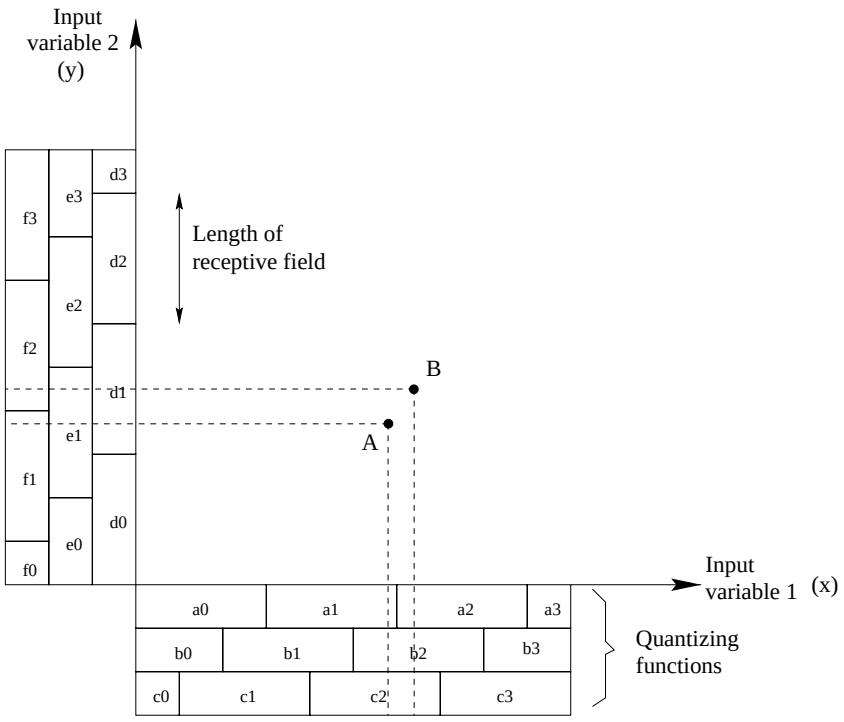


Figure 5.14: Example of CMAC coarse coding.

5.8.3 Local function approximators

Global learning systems (e.g., an MLP using Backpropagation) can achieve greater generalization than local function approximators from each training pattern due to the global adaptation. However, local function approximators like RBF neural networks and CMAC models can learn faster and more efficiently since new knowledge can be incorporated by local modifications in the network. One difference between RBF networks and CMAC is that in CMAC the Q-value estimated by the indexed weights (Subsection 5.8.1) is constant across an entire region, while in the RBF the Q-value is maximal at the center of the receptive field and then drops gradually away from the center [111]. Moreover, in CMAC, the partition of the input state space is fixed *a-priori*, while in RBF, it is possible to adapt the center and the size of the receptive fields [10] (see also Subsection 4.5.1). Recent approaches have tried to merge the best features of both RBF and CMAC models as presented in the following section and in [54].

5.8.4 Clustering techniques

Clustering techniques and in particular unsupervised learning clustering algorithms are very useful to adaptively partition the input space of a system. When a specific region of the input space (i.e., the input of features) presents sufficiently similar characteristics, it is considered as a category, and a unique table entry is associated to it. Adaptive clustering techniques are particularly useful when it is difficult to determine *a-priori* the partition of the input space (i.e., by defining tiles, groups of states, or receptive fields).

Most unsupervised clustering methods form clusters or categories based on the den-

sity distribution of the features in the input space and determine the similarity of two features based on a measure of distance in the input space (Chapter 4).

Although many clustering techniques exist in the literature, we were particularly interested in the adaptation of such techniques for the development of an online clustering algorithm, motivated by the possibility of a digital implementation. We have thus developed a neural architecture dubbed FAST (Flexible Adaptable-Size Topology) that dynamically partitions the input space of a system by setting up a variable number of adaptable receptive fields (i.e., with modifiable centers and sizes). In the following section, we will describe how FAST is able to dynamically partition the input space of a non trivial control problem (the inverted pendulum problem). The resulting dynamic coarse coding allow the use of a look-up table implementation of the adaptive heuristic critic (AHC) algorithm to solve such problem by reinforcement learning. It should also be noted that FAST (Chapter 4) is able to dynamically localize overlapping receptive fields in the input space of a system, which can be considered a CMAC-like adaptive coarse coding technique. Moreover, if the activation of a FAST neuron is made inversely proportional to the distance between the input vector and the receptive field center, a FAST network will merge the characteristics of the RBF and the CMAC techniques.

5.9 The inverted pendulum problem

The inverted pendulum (Figure 5.15) is a classic example of an inherently unstable system. This problem has often been used to test new approaches to learning control (from the early work of Widrow and Smith in the 1960s [230] to recent studies, such as Lin and Lin's [115]). The system consists of a pendulum hinged on the top of a cart, which is free to move within the limits of a horizontal track ($-2.5m < x < 2.5m$). The pendulum is constrained to move in the vertical plane, perpendicular to the cart's motion (i.e., it cannot move sideways). The state of the system is determined by the position of the cart (x), the speed of the cart (dx/dt), the angle of the pendulum (θ), and the angular velocity of the pendulum (dθ/dt). The possible actions to attempt to maintain it in equilibrium are *push to the right with a force of +10 Newtons* and *push to the right with a force of -10 Newtons* (i.e., push to the left with a force of +10 Newtons).

In our simulations, the inverted pendulum was modeled by a set of differential equations as in [23]. However, it is assumed that the neurocontroller does not have any explicit information about the dynamics of the system and the only information regarding the goal is through an environmental reinforcement parameter, r, defined as follows:

$$r = \begin{cases} 0 & if -2.5m < x < 2.5m \\ & and -12^o < \theta < 12^o \\ -1 & Otherwise. \end{cases}$$

Intuitively, r indicates whether the control system is acting in a "good" manner (0) or not (-1). The initial state of the system is assumed to be $x = 0$, $dx/dt = 0$, $\theta = 0$, and $d\theta/dt = 0$.

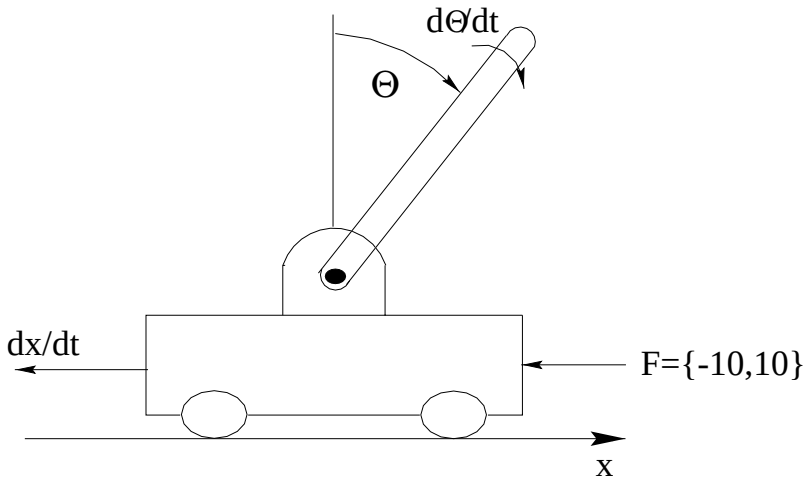


Figure 5.15: Inverted Pendulum System. The state of the system is determined by the position of the cart (x), the speed of the cart (dx/dt), the angle of the pendulum (θ), and the angular velocity of the pendulum (dθ/dt). The objective is to maintain the pendulum in equilibrium by pushing the cart with a force F to the left or to the right, while keeping the cart within the limits of a horizontal track.

5.9.1 The boxes-AHC neurocontroller

Since the state set of the inverted pendulum problem is continuous, the solution to the problem includes the definition of a discrete representation of the input space. In the early work of Michie and Chambers [136], a prespecified number of regions (originally called *boxes*) were determined to enable a look-up table implementation of the solution (Figure 5.16).

$$
\begin{array}{c}
x : \pm 0.8m, \pm 2.4m, \\
\theta : \pm 1^o, \pm 6^o, \pm 12^o, \\
dx/dt : \pm 0.5m/s, \pm \infty m/s, \\
d\theta/dt : \pm 50^o/s, \pm \infty^o/s.
\end{array}
$$

Figure 5.16: Discrete representation of the input space of the inverted pendulum problem.

More recently, the well-known paper of Barto, Sutton, and Anderson [23] used the same discrete representation and used the adaptive heuristic critic algorithm (AHC) to learn how to solve the inverted pendulum problem by interaction.

The AHC neurons compute the weighted sum of their inputs and compute their activation as follows :

$$
y(x) = \begin{cases} -1 & if\ f(\sum_{i=1}^{n} w_i(t)x_i(t)) < rnd \\ +1 & otherwise \end{cases} \tag{5.10}
$$

where the weights w_i indicate the preference for taking an action, f is the sigmoid

function, and *rnd* is a uniformly distributed random number in the range $(0, 1)$. The output action of the AHC neurocontroller is a force of +10 Newtons (if $y(t) = +1$) or -10 Newtons (if $y(t) = -1$) applied to the cart.

The weight modification rule is as follows :

$$w_i(t+1) = w_i(t) + \alpha r(t) e_i(t) \ , \tag{5.11}$$

where α is a positive constant determining the rate of change of w_i, $r(t)$ is the external reinforcement parameter, and $e_i(t)$ is an eligibility trace.
In [23], the eligibility is computed as :

$$e_i(t+1) = \delta e_i(t) + (1-\delta) y(t) x_i(t) \ , \tag{5.12}$$

where $\delta (0 \leq \delta < 1)$ determines the decay rate of e_i.
The predicted reinforcement $p(t)$ is also computed as a weighted sum of the inputs:

$$p(t) = \sum_{i=1}^{n} v_i(t) x_i(t) \tag{5.13}$$

The v_i weights (i.e., the state values) are updated as follows:

$$v_i(t+1) = v_i(t) + \beta [r(t) + \gamma p(t) - p(t-1)] \overline{x}_i(t) \ , \tag{5.14}$$

where β is a positive constant determining the rate of change of v_i $(\gamma, 0 < \gamma \leq 1)$ is a constant "discount factor", and $\overline{x}_i$ is a trace which acts much like the eligibility for the w_i weights. $\overline{x}_i$ is computed as follows:

$$\overline{x}_i(t+1) = \lambda \overline{x}_i(t) + (1-\lambda) x_i(t) \ , \tag{5.15}$$

where $\lambda (0 \leq \lambda < 1)$ determines the decay rate of $\overline{x}_i$.

5.9.2 Function approximation by adaptive clustering

An *a-priori* partition of the input space like the one presented in last section is not always feasible and often requires an in depth analysis of the system. A different approach consists of using a clustering algorithm (or more complex function approximation techniques, as described in Section 5.8) to partition the state space of the system.

We have used the FAST unsupervised clustering network (Section 4) to dynamically adapt the structure of the neurocontroller. For this task, a 3D-input FAST network clusters the input space variables of the system being controlled (Figure 5.17) and activates or deactivates neurons in the clustering and reinforcement modules.

In our system, dynamic clustering of the input space corresponds to adaptive partitioning of the state space of the system being controlled (i.e., the inverted pendulum). Instead of pre-determining a set of regions (boxes) in the state space of the system, the unsupervised algorithm dynamically finds such regions, by incrementally activating neurons and dynamically adapting their boundaries [154]. Note that, once a FAST neuron is added/deleted, the corresponding look-up tables in the reinforcement module increase/decrease in size. Therefore, a FAST-AHC system can be considered as an ontogenic reinforcement-based neurocontroller (i.e., a structure-adaptable neurocontroller).

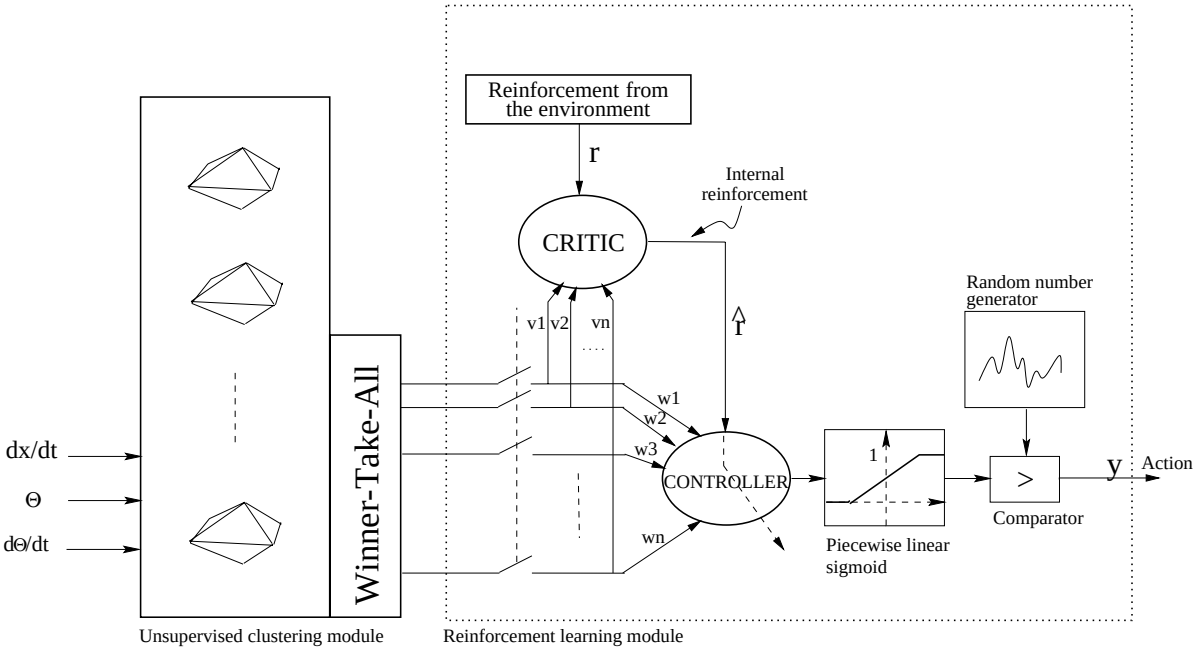


Figure 5.17: Block diagram of the neurocontroller. The first module is a 3D-input FAST neural network, which partitions the input state into octahedrons, and the second module is a reinforcement learning neural network.

5.9.3 The FAST-AHC neurocontroller

Our most complex system to date consists of 10 FAST neurons in the unsupervised clustering module, a Winner-Take-All operator, and an adaptive heuristic critic (AHC) module (Figure 5.17).

Although the constructive algorithm determines the appropriate size of the FAST neural network, i.e., the number of regions in the state space of the system, the shapes of the clusters (hyper-spheres for Euclidean distance and hyper-octahedrons for Manhattan distance) require a high number of neurons to solve the problem. We therefore limited the number of neurons in the algorithm and added a winner-take-all operator to associate every non-classified input vector with the nearest cluster in the state space. After several tests (Table 5.6), we decided to constrain the size of the network to 10 neurons. In our system, clusters correspond to partitions in the 3D state space $\{\mathrm{dx/dt}, \theta, d\theta/dt\}$. The position of the cart (x) was ignored for clustering purposes so as to minimize the hardware implementation. The values of the parameters in the learning rules were $T_{ini} = 0.9875, T_{min} = 0.5, \gamma = 0.0078, L_r = 0.7969, P_{rmin} = 0.0938, \eta = 0.0469$ (See Figure 4.10). The AHC module was implemented using look-up tables, and the sigmoid transfer function was approximated by a piece-wise linear function. See Section 5.10 for more details.

5.9.4 Results

Our results show that an adaptable-size clustering algorithm can be successfully applied to properly partition the input space of the inverted pendulum system. Although this is

Number of Neurons	Percentage of Success (%)
4	76 %
5	72 %
6	92 %
7	93 %
8	97 %
9	98 %
10	100 %

Table 5.6: Percentage of success for different FAST network sizes. The success rate is determined after 100 tests, and the system is considered to have failed a test if it needs more than 1000 trials to balance the inverted pendulum during one simulated hour.

		Adaptive heuristic critic (AHC)	FAST10-AHC
1	Mean	115.4	262.5
	Best	30	15
	Worst	946	682
2	Mean	13,094s	475.9s
	Best	108.6s	6.4s
	Worst	493,876s	4,884s
3	Failures	1	0

Table 5.7: (1) Inverted pendulum balance attempts. (2) Simulated time needed for learning, in seconds. (3) Number of failures.

a well-known system and it has been solved by numerous methods, including adaptive control, supervised neural networks, fuzzy logic control, and genetic algorithms, we are not interested in the inverted pendulum problem *per se*, but in its use as a testbed for the development of dedicated digital hardware for neurocontrollers.

The success rate was determined after 100 tests: the system failed during a test when it needed more than 1000 trials to balance the inverted pendulum during one simulated hour. With a sampling of 0.02 s^{-1}, one simulated hour corresponds to 180.000 learning steps. Further simulations have shown that the size of the network plays a critic role in the input space partition. As shown on Table 5.7, a 10-neuron FAST network led to a success rate of 100% compared to 99% of the original neuron-like model of Barto, Sutton, and Anderson [23].

We found that our system exhibits better performance than Barto, Sutton, and Anderson's original system also when considering other different criteria. Figure 5.18 shows the total number of learning steps (in the 100 tests) required by the AHC algorithm and a configuration of 8 FAST neurons for our system before learning to balance the

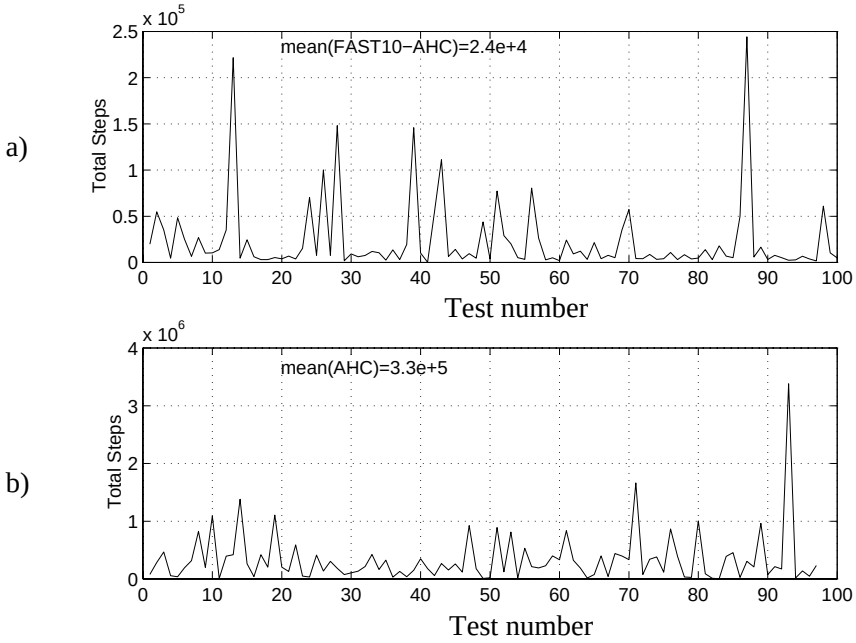


Figure 5.18: Performance of a) our system and of b) the original adaptive heuristic critic (AHC) system in 100 different tests of the inverted pendulum problem. The Figure does not show the failure case and two extreme cases (7.5e+6 and 24.7e+6 steps) of the AHC algorithm. Notice that the number of iterations are in different scales.

pendulum in 180.000 steps (1 step = 0.02 seconds). Table 5.7 shows that our system is more than 25 times faster (on average) than the adaptive heuristic critic algorithm.

Table 5.7 also shows the minimum (best) and maximum (worst) number of trials for learning, and the number of failures (out of the 100 tests) for both neurocontrollers. Table 5.8 shows the average of trials and the total number of steps required to learn with the adaptive heuristic critic (AHC) and with several configurations of our system.

It should be noted that, although on average the AHC algorithm needs fewer attempts to learn to balance the pendulum during one simulated hour, it requires a much longer time than our system. This is due, in part, to the adaptable partition of the state space and to the fact that the external reinforcement is more informative when the system fails than when it does not. The significant performance measure for the inverted pendulum problem is learning time, rather than number of failures. However, this latter measure can be of importance in other classes of problems, e.g., in obstacle-avoidance tasks for autonomous mobile robots, where a larger number of failures can cause physical damage to the system.

5.10 FPGA implementation of neuron-like adaptive elements

The AHC network of neuron-like adaptive elements and the FAST unsupervised clustering network were implemented on a custom machine based on field programmable gate

	Mean (Trials)	Std_dev (Trials)	Mean (Steps)	Std_dev (Steps)	% Success
AHC	115.4	136.7	654,700	2.5e+6	99%
FAST8-AHC	163.7	93.5	19,339	4.4e+4	97%
FAST9-AHC	200.5	98.6	15,160	2.7e+4	98%
FAST10-AHC	262.5	128.1	23,798	4.1e+4	100%

Table 5.8: Average of trials and total number of steps for learning with: (a) adaptive heuristic critic, (b) 8 FAST neurons and AHC, (c) 9 FAST neurons and AHC, and (d) 10 FAST neurons and AHC.

array (FPGA) devices, developed in our laboratory. It is composed of a 68EN360 microcontroller with Ethernet 10Base-T interface, two Altera Flex 10K100 FPGA chips (each with approximately 100,000 equivalent gates [8]), 4MB of DRAM accessible from the FPGAs, 8MB of DRAM accessible from the microcontroller, RS232 and Mubus interfaces, and a 25 MHz clock. We used one of the Altera Flex 10K100 FPGAs to implement four FAST neurons and an adaptive heuristic critic (AHC) network with 8 synapses (the second FPGA was not available at the time of this implementation).

5.10.1 The neuron-like adaptive elements

We have implemented the AHC algorithm to control a simplified version of the well known inverted pendulum problem [23]. As described in section 2 each neuron-like element consist of two look-up tables: the ASE element stores w_i and e_i, and the ACE element stores v_i and $\overline{x}_i$. Figure 5.19 shows the implementation of the look-up tables and the arithmetic operations needed for their update. There are as many *synaptic* blocks as synapses on the neurons. Our implementation currently supports 8-bit precision and there are up to 4 synapses per neuron. All variables in the algorithm are coded using 8 bits but the range of values they represent are different: $w_i \in (-8, 8)$, $v_i \in (-1, 0)$, $\overline{x}_i \in (0, 1)$, and $e_i \in (-1, 1)$. The values of the constants in the algorithm are also coded with 8-bit precision: $\alpha = 127$, $\beta = 0.5$, $\gamma = 0.9531$, $\lambda_w = 0.875$, and $\lambda_v = 0.7969$.

Each synaptic block includes two 8-bit adders and a single 8-bit shift-add multiplier. Additions occur in parallel, but the multiplications involved in the learning phase (six) are executed sequentially.

Two 8-bit buses are provided to let the synaptic blocks communicate with an *activation* block, shown in Figure 5.20. This latter block computes the output activation of the ACE and ASE neurons. It contains four 8-bit adders, a single 8-bit shift-add multiplier, a comparator, a cellular automata-based random number generator [90], and a linear piecewise approximation of a sigmoid activation function (Figure 5.21).

In our system, the output action is a function of the inputs and the weights w_i of the actor. The steep of the sigmoid function determines the exploration-exploitation ratio: if the sigmoid function is too close to a step function, the system will exploit most of the time, but, if the sigmoid function is not too steep, the system will explore too much. We have experimentally found that the piece-wise linear approximation of

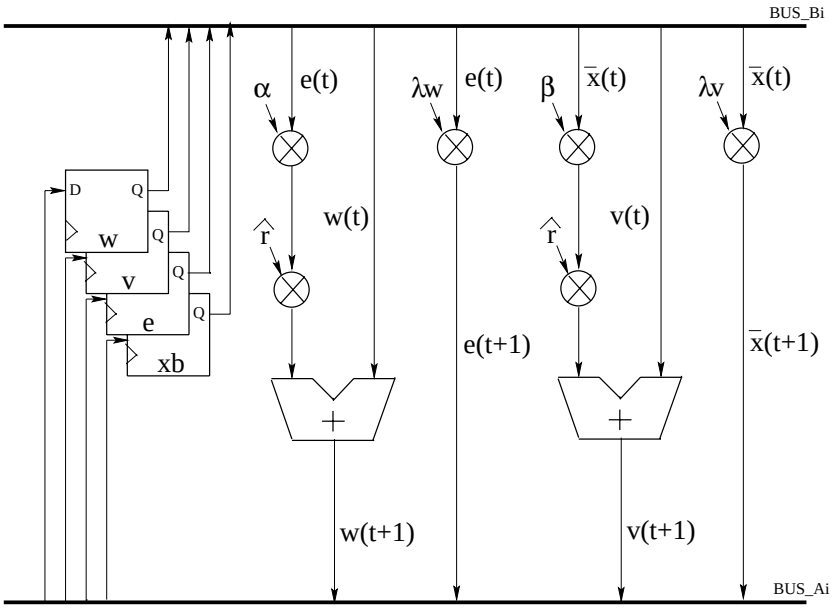


Figure 5.19: Digital implementation of a *synaptic block*

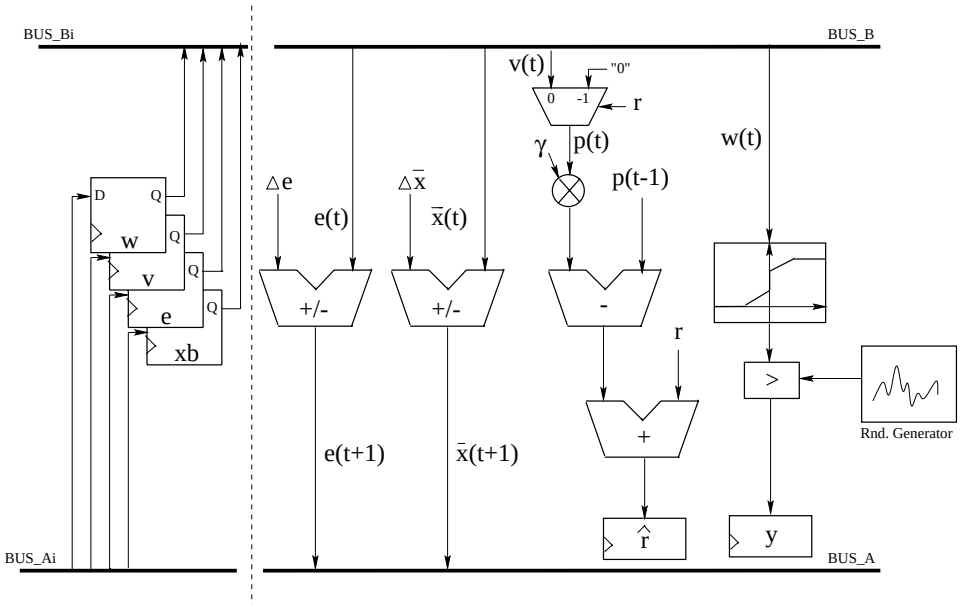
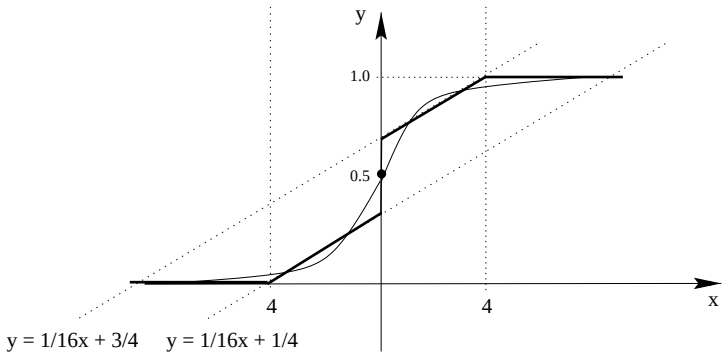


Figure 5.20: Digital implementation of the *activation block*



$$y(x) = \begin{cases} 1 & if\ x \geq 4\ , \\ 0.0625x + 0.75 & if\ 0 < x < 4\ , \\ 0.5 & if\ x = 0\ , \\ 0.0625x + 0.25 & if\ -4 < x < 0\ , \\ 0 & if\ x \leq -4 \end{cases} \tag{5.16}$$

Figure 5.21: Piece-wise linear approximation of a sigmoidal activation function.

Figure 5.21 resulted in a good tradeoff. Note that the values of the constants of the linear approximation are powers of two, and that the sigmoid is thus implemented with shift and add operations.

The sequencer of the system is composed of two finite state machines (FSM) which control the sequential operation of the system. The first state machine handles the synchronization of weight and trace updates. The second state machine handles the output activation calculation and the reinforcement prediction, and freezes the system while applying the resulting action on the environment and receiving the external reinforcement evaluation of such action.

5.10.2 The neurocontroller

The neurocontroller system consists of an AHC network, a FAST network, and a memory interface block. This block consists of 18 8-bit registers used to map every input to the neurocontroller and of a finite state machine that handles the communication with the 68EN360 microcontroller, where a software simulator of the inverted pendulum dynamics runs. The communication between the hardware device and the microcontroller occurs via a shared dynamic RAM memory. A polling subroutine runs on both the 68EN360 microcontroller and the FPGA (where it is implemented as a state machine) in order to generate the read/write control signals and to receive/send data. A detailed description of the design and hardware implementation of the FAST network can be found in [157].

The worst-case neurocontroller processing time per input vector is only 152 clock cycles. The actual time depends on the number of multiplications to be performed during learning. However, the access time of the DRAM memory (seven 25MHz clock cycles) limits us to about 7500 input vectors per second. In the next chapter we will describe a new neurocontroller, based on the FAST architecture and SARSA-learning instead of

AHC, intended for the control of a mobile robot.

5.11 Summary

Adaptive organisms have to deal with a perpetually changing environment. Thus, in order to survive, they must be able to predict future events by generalizing the consequences of their behavioral responses to similar situations experimented in the past.

Temporal-difference methods are a computational approach to learn to predict by trial-and-error learning. Such methods belong to a wider class of adaptive techniques called reinforcement learning algorithms, where a system learns by interaction with the environment, without the need of an external supervisor. Applications of these learning algorithms arise in sequential-decision processes, such as the development of optimal strategies for gaming and control.

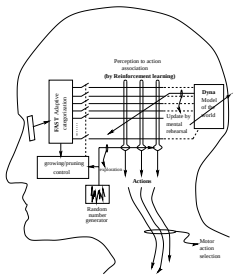
Reinforcement learning problems are usually modeled as Markov decision processes (MDPs), even though they can also be used in non-Markovian tasks. Reinforcement learning systems have to deal with the *temporal credit assignment* problem, i.e., how to punish or reward actions when they have long-lasting effects, and the *exploration-exploitation dilemma*, i.e., how to decide when to try new actions (to explore) in order to discover better action selections for the future [200]. A series of learning algorithms have been developed to attempt to solve those two problems. In particular, we have studied the most commonly used reinforcement learning methods: Q-learning, SARSA learning, and the adaptive heuristic critic (AHC) learning algorithms. We have presented the problem of generalization in complex problems (i.e., problems with large state spaces), and two approaches that optimize TD-learning, namely TD(λ), a TD-learning methods with memorization of the occurrence of events (the so-called eligibility traces), and Dyna, an architecture that integrates TD-learning and planning methods.

We have used these kind of algorithms to learn strategies in the game of Blackjack, to solve maze tasks and to solve a non-trivial control problem (the inverted pendulum problem). Moreover, we compared the performance of such algorithms in Markovian and non-Markovian maze tasks and studied their possibility of a digital implementation to develop a hardware-friendly neurocontroller architecture.

Finally, we used high-density programmable logic devices (FPGAs) to implement a neurocontroller composed of an unsupervised clustering algorithm (FAST) and the adaptive heuristic critic (AHC) learning. The successful use of this neurocontroller with the inverted pendulum problem encouraged us to develop a new neurocontroller for autonomous mobile robot navigation, which will be described in the next Chapter.

Chapter 6

A Neurocontroller Architecture for Autonomous Robots



"Intelligence is what you use when you don't know what to do."

-J. Piaget

Chess has always been regarded as a game of intelligence. Humans have tried to build machines capable of playing chess since at least the eighteenth century, when Baron Wolfgang von Kempelen devised a chess-playing mechanical automaton [92]. More recently, Shannon, Turing, and von Neumann were also interested in the idea of devising a chess-playing machine by programming a computer. Turing developed a move-generating and position-evaluating program that he tested by hand, whereas von Neumann and Morgenstern devised a so-called *minimax* algorithm to calculate the best move. The seminal work about chess-playing machines was then published by Shannon in 1950 [189].

In 1965 the Russian mathematician Alexander Kronrod said: "Chess is the Drosophila of artificial intelligence" [126]. Chess, and games in general, became the testbed for intelligent machines. In particular, the discrete nature of most games eliminate the cumbersome analog-to-digital conversion necessary to translate our physical environment for manipulating and sensing machines. In Samuel's words: "A game provides a convenient vehicle for the study of machine learning systems as contrasted with a problem taken from life, since many of the complications of detail are removed" [176]. Samuel implemented the first game learning program to play checkers [176].

On May 11, 1997, Deep(er) Blue, a machine devised by a group of IBM defeated grand-master G. Kasparov, arguably the best chess human player in history, after two wins, a loss, and three draws. Even though this may have proven the feasibility of im-

plementing an intelligent machine, it was soon apparent that Deep(er) Blue was nothing but a very powerful computing machine with the ability to evaluate 200 million moves per second, that is, of "packing 380 years of human thought into 3 minutes" [78].

The removal of details, or "complications", in a game learning problem was perfect for classic artificial intelligence techniques. Nevertheless, such methods soon revealed their limited performance, particularly when dealing with problems other than games, as is the case of robot control.

Indeed, once a system has to interact with the real world, the truly difficult problems of intelligence come to the fore, and classical AI methods do not work very well. Therefore, a new kind of problem was require to provide a new testbed for the ideas of intelligence. Such a problem was found in autonomous robot control. In Brooks' words:

> "To really test ideas of intelligence it is important to build complete agents which operate in dynamic environments using real sensors" [31].

As a consequence, some researchers feel that the ultimate goal of artificial intelligence research is to develop autonomous robots.

Early work on robot control was based on a *sense-model-plan-act* (SMPA) framework, as described by Brooks [30]. The system attempted to recover as much of the state of the environment as possible, operated on the internal representation of the state using general planning and reasoning algorithms, and chose a sequence of control actions to implement the selected plan. However, the complexity of designing such architectures led researchers to investigate simpler solutions. Around 1984, some researchers began to consider the control of robots using simple combinatorial circuits plus a small timing circuitry [30]. The result was the development of a new approach called *behavior-based* robotics or *reactive* robotics [14, 15, 30].

A behavior-based robot uses low-level primitive processes relating sensations and actions (behaviors), hoping that complex physical behavior will emerge through interaction with the environment. It uses the environment as its own model, continuously referring to its sensor readings rather than to an internal world model. Such a system is thus said to be *reactive* (the intelligence of the system emerges from its interactions with its environment), *situated* (the robot does not deal with abstract descriptions of the world, but interacts directly with the environment), and *embodied* (the robot has a body, experiences the world directly, and its actions have immediate feedback on its own sensations) [30].

One of the problems with behavior-based robots is that the component modules (i.e., the behaviors) have to be programmed by a human designer [127]. Nevertheless, the behaviors could be learned through *trial-and-error* coupled with reward/punishment mechanism, i.e., by reinforcement learning.

Reinforcement learning has provided algorithms that use evaluation signals to guide a search for improved behaviors. Moreover, these algorithms have been shown to approach the optimal plans that are derived (with more computational effort) from explicit planning algorithms [102].

The learning of behaviors in a reactive robot is a challenging problem. We are particularly interested in the development of a neurocontroller architecture for the autonomous

navigation problem. Our neurocontroller is based on an unsupervised clustering module, a reinforcement learning module, and a planning module, as will be detailed in the present chapter. First, we start by introducing the problem of mobile robot navigation learning in Section 6.1. In Section 6.2 we introduce the neurocontroller architecture we have developed. Section 6.3 describes the digital implementation of the neurocontroller and presents some tests and results. Section 6.4 presents an extension of the neurocontroller architecture with the introduction of short-term memories to disambiguate identical sensory inputs observed in different states.

6.1 Autonomous robot navigation

We can distinguish two types of approaches to navigation-learning: *skill-based* learning and *model-based* learning [202]. In the skill-based learning approach, the robot learns the skills required to achieve a specific goal, such as homing in on a target or going around in circles, while in the model-based approach, a process of planning using an internal model of the world allows the robot to adapt to different goals.

6.1.1 The navigation task

The general framework for autonomous navigation assumes the following conditions:

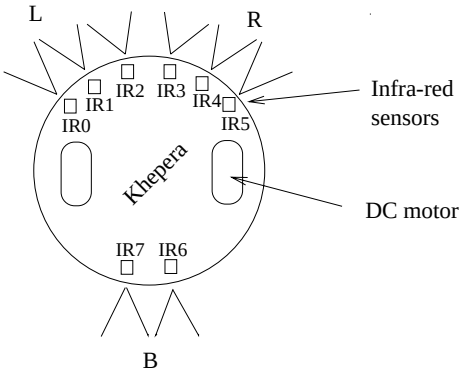
- The robot does not have access to its global position within the workspace, it should navigate based exclusively on its local sensory inputs.
- There are no explicit landmarks accessible to the robot.
- No *a-priori* knowledge of the workspace geometry is available.
- The robot has to be robust with respect to transient disturbances, including noise and geometrical changes in the workspace.

6.1.2 The experimental setup

We have used the Khepera mobile robot developed at the Microcomputing Laboratory, Swiss Federal Institute of Technology in Lausanne (EPFL) [139]. It has a circular shape (Figure 6.1), with a diameter of 55 mm, a height of 30 mm, and a weight of 70 g. It has two wheels controlled by two DC motors, and eight infrared sensors working in active or passive mode. In active mode they emit infrared light and measure the quantity of reflected light (i.e., they act of sensors of proximity), while in passive mode they simply measure the infrared component of ambient light giving a rough estimate of light intensity [61]. The mobile robot contains a Motorola 68331 microcontroller, 256 Kbytes of RAM, 512 Kbytes of ROM, and a RS-232 serial link at 9600, 19200, or 38400 Baud. On-board rechargeable batteries provides around 30 minutes of autonomy. The Khepera robot is modular: several extension turrets with CCD cameras, linear vision systems, grippers, artificial retinas, auditory systems, etc., can be plugged-in to extend

its capabilities. An extension bus and its corresponding control signals also allow the user to design his/her own extension turrets, such as for example, an FPGA board.

The microcontroller provides infra-red readings with a 10-bit precision. In our experiments, we have used three 8-bit values as sensor readings: L (left front), R (right front) and B (back), corresponding to a preprocessing of the eight infra-red sensor signals:



$$L = \max(IR0, IR1, IR2),$$

$$R = \max(IR3, IR4, IR5),$$

$$B = \max(IR6, IR7)$$

These 8-bit values encode a real number in the range [0 1). The speed of the two motors can be set in steps of 8 mm/s. The maximum speed is around 1 m/s [103]. In our experiments, we determined four possible actions *go forward*, *go right*, *go left*, and *go backwards* with a fixed speed of 40mm/s. The first three actions are associated to particular situations during learning, while the last action is only used as part of a pre-wired *basic reflex* codified as a simple reactive behavior: follow the direction of the least-activated sensors during 10 times 10 time steps. This basic reflex is activated when one of the infra-red sensor exceeds a certain threshold value.

The robot is punished (it receives a reinforcement of '-1') when it crashes or turns on itself. Otherwise, it receives '0' as reinforcement. The robot holds two variables dl and dr that indicate if the system moves left or right too much. These variables are updated when the robot moves as follows:

$$dl = \Delta_l + (1 - |\Delta_l|) * dl \ , \tag{6.1}$$

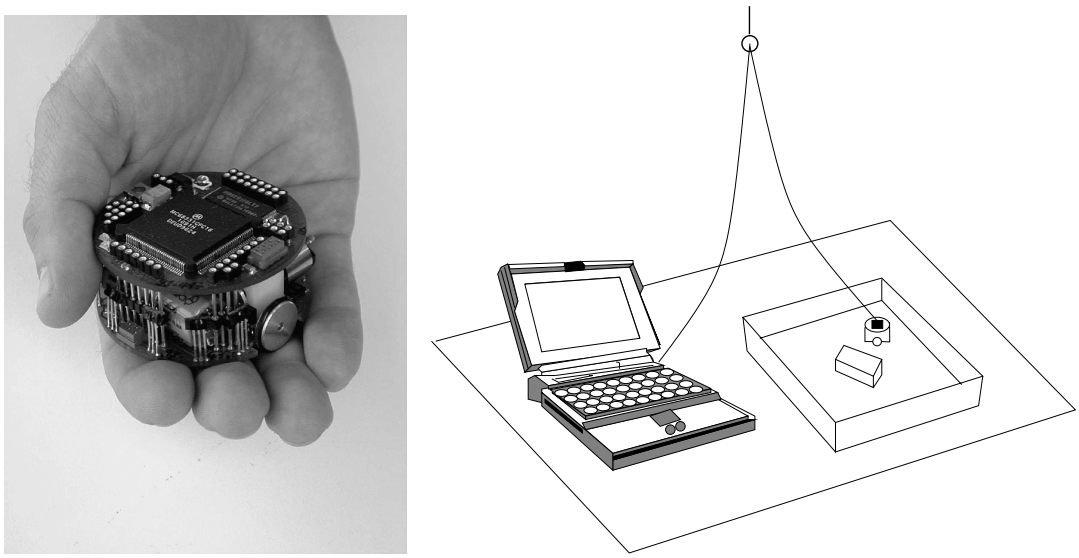
$$dr = \Delta_r + (1 - |\Delta_r|) * dr \ , \tag{6.2}$$

where Δ_r and Δ_l are defined as follows:

$$\begin{aligned}
\Delta_r = 0.01, \Delta_l = 0.01, \quad & if\ action = go\ forward, \\
\Delta_r = -0.01, \Delta_l = 0.01, \quad & if\ action = turn\ left, \\
\Delta_r = 0.01, \Delta_l = -0.01, \quad & if\ action = turn\ right, \\
\Delta_r = -0.01, \Delta_l = -0.01, \quad & if\ action = go\ backwards
\end{aligned}$$

The robot is thus reinforced as follows:

$$r = \begin{cases} & if\ (L > 0.7)\ or\ (R > 0.7)\ or\ (B > 0.7), \\ -1 & if\ (dl - dr) > 1.5)\ or\ (dl - dr) < -1.5), \\ & if\ (dl + dr) < -0.6, \\ 0 & otherwise \end{cases}$$



a) b)

Figure 6.1: Experimental setup. a) The Khepera mobile robot. [Photo by André Badertscher] b) Khepera in its workspace connected to a host computer for power supply and control or visualization.

The workspace utilized to explore the navigation-learning task with the Khepera mobile robot, was a 40 cm $\times$ 40 cm square-box made of wood, shown in Figure 6.2. A set of bricks of wood were used to modify the contour and the obstacle in the center of the arena.

6.2 Learning in autonomous robots

Programming an autonomous robot so that it reliably acts in an unknown or a dynamic environment is a difficult task, due to the lack of information during programming, the dynamic nature of the environment, and the inherent noise in the robot's sensors and actuators [52].

A standard approach to the control of autonomous robots is that of *reactive* systems or *behavior-based* robotics. However, such approaches may not generate good solutions to the navigation problem since the control system reacts as a function of the sensor's readings and the robot's perception is limited. An interesting approach is to add learning capabilities to reactive systems [137].

6.2.1 Related work

The development of learning techniques for mobile autonomous robots constitutes a major field of research. A lot of work has been done running simulated robots in simulated environments, though sometimes good results obtained in such a way do not hold true in the real world [52]. More recent research plans to use simulation to speed up the initial adaptation of the controller [61]. *Evolutionary techniques* have been successfully used

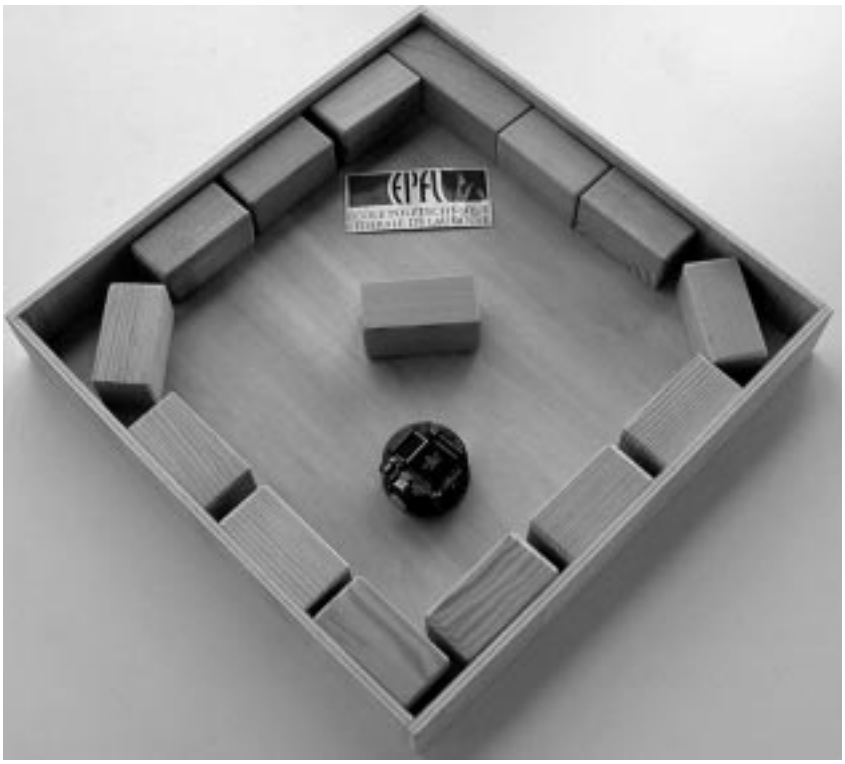


Figure 6.2: The navigation workspace. [Photo by André Badertscher]

to adapt a robot controller while considerably reducing human design, even though the duration of the experiments is rather long [61]. Our approach tries to speed up adaptation by means of learning by interaction instead of evolution, and is closely related to the works of Zrehen [243] based on a probabilistic self-organizing map, Kröse et al. [112], Bruske et al. [32], and C. Scheier [181], based on ontogenic neural structures of RBF units, and to Millan's incremental learning approach [137], based on Alpaydin's GAL supervised learning model [6]. However, such works have not considered the possibility of a digital implementation, which is a key motivation in our research.

6.2.2 Hardware for learning robots

From the hardware point of view, several previous works have developed extensions boards for the Khepera mobile robot. Lund et al. [123] developed an auditory system used in robotic experiments in cricket phonotaxis, and the robot has been tested with real male crickets of the species *Gryllus bimaculatus*. When the crickets sing the calling song, the robot (that models a female cricket) responds by turning and moving directly toward the crickets. López de Meneses [122] developed an extension turret with a panoramic, linear artificial retina developed by the Centre Suisse d'Electronique et de Microtechnique (CSEM) in Neuchâtel, Switzerland. Thompson [207] developed an FPGA-based extension turret to run evolvable hardware experiments (e.g., the evolutionary design of electronic control circuits) with the Khepera robot.

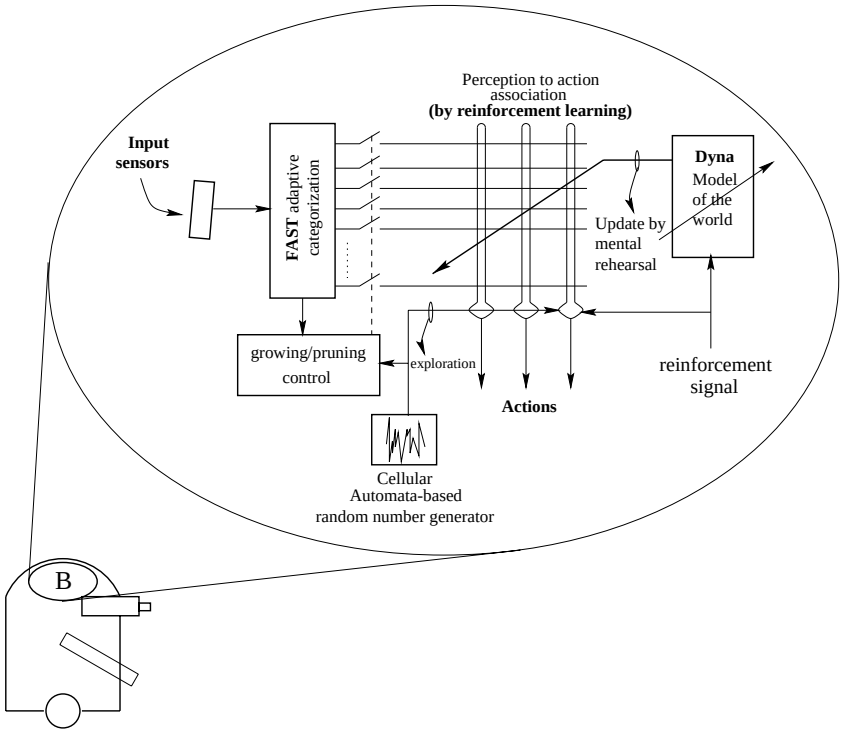


Figure 6.3: Neurocontroller architecture. It is composed of an adaptive categorization module and a reinforcement learning module. The first module is implemented by the FAST neural architecture, which uses a growing/pruning control block and a cellular-automata-based random number generator. The second module implements a Dyna-SARSA learning algorithm which dynamically creates a model of the environment to integrate TD-learning and planning techniques.

6.3 Neurocontroller architecture for autonomous robot navigation

An autonomous robot needs not be given all the details of the environment in which it is going to operate: it will acquire them by direct interaction and extrapolation [52]. To achieve this, the robot's neurocontroller has to be able to autonomously categorize the input data it receives from its environment, to handle with the *stability-plasticity* trade-off (i.e., how can it preserve what it has previously learned, while continuing to incorporate new knowledge), and, finally, to generalize between similar situations and develop a proper policy for action selection, based on an evaluative reinforcement signal. Figure 6.3 presents a neurocontroller architecture with the above mentioned capabilities. It is composed, essentially, of an adaptive categorization module and a reinforcement learning module.

6.3.1 Adaptive categorization module

To address the problem of adaptive categorization, we used the Flexible Adaptable-Size Topology neural network described in this thesis (Chapter 4). While FAST is not the first neural network based on adaptable topology, it is special in that it does not require intensive computation to learn and reconfigure its structure, and can thus exist as an on-line learning stand-alone machine that can be used, for example, in real-time control applications, such as robot navigation. In our approach, FAST dynamically categorizes the 3-dimensional infra-red sensor signals (L,R,B), serving as a sparse-coarse-coded function approximator of the input state space (Section 5.8), providing generalization, and, at the same time, dealing with the stability-plasticity dilemma (by locally modifying the weights). In particular, it should be noted that our system has to be able to handle the constantly changing activation of its three 8-bit "eyes", which correspond to 24 binary receptors that can produce 2^{24} (approximately 1.7×10^7) different combinations of ones and zeros. Although much simplified compared to the 10^8 receptors of each human eye or the 256×256 receptors of a CCD camera, this task allows the study of adaptive categorization in learning robots.

Our system has been design to support 8-bit computation, and a maximum number of 40 categories was selected for this application. The values of the FAST learning parameters were initialized as follows:

T_{ini}	T_{min}	γ	L_r	Pr_{max}	Pr_{min}	η
0.1	0.01	0.01	0.2	1.0	0.9	0.01

In particular, it should be noted that these parameters determine a very slow decrease in the FAST thresholds (γ) and a very low rate of pruning (η). The γ value was determined on an empirical basis, while the low pruning rate was intentional: during the first learning steps, the first activated neurons are selected repeatedly and, if the sensitivity regions of two or more of them overlap, pruning is very likely to occur (even if Pr, the probability of not being pruned, is very high). Further discussion about the dynamics of this part of the neurocontroller is presented in Subsection 6.3.3.

6.3.2 Reinforcement learning module

To make a *reactive* system learn by interacting with the environment, we used reinforcement learning techniques [200], which allow the system to selectively retain the actions that maximize the received reward over time. The reinforcement learning module associates a particular action (by means of a state-value function) to each environmental situation, determined by the learning system's sensors and the adaptive categorization module.

In Chapter 5 we considered several reinforcement learning techniques and studied their feasibility as a digital implementation. Our first implementation was the adaptive heuristic critic (AHC) algorithm (Section 5.10). However, SARSA and Q-learning are computationally less intensive, because the actor and the critic modules are implemented using the same set of values. Q-learning has been widely used and offers a mathematical proof of convergence. Nevertheless, such proof is not valid for non-Markovian tasks, which is the case in the problem of robot navigation. We chose SARSA because the state-action value update is simpler than in Q-learning, where an additional maximum operation is needed. We tested SARSA(λ) and Dyna-SARSA in our robot navigation task, but focused our studies on the Dyna version because it is computationally less intensive than SARSA(λ), and it is well suited for a digital implementation.

In particular, we implemented a Dyna-SARSA adaptation for non-Markov Decision Processes (Subsection 5.6.2). The Dyna framework profits of a dynamically constructed model of the environment to update its behavior offline. In tasks where the cost of interacting with the world is high, it is an efficient way to produce a process of *mental rehearsal* of what has been previously experienced, in order to accelerate learning (although, it is difficult to say if this learning process is biologically plausible, a similar approach appears to be used by athletes to enhance their performance). This may be misleading in non-deterministic environments, but a proper rate of offline-online updates and value function approximation techniques can lead to good results, as will be shown in the next subsection. We were particularly interested in this approach because a digital neurocontroller or a digital neural coprocessor can be realized to allow the system to rapidly generate hundreds of synthetic interactions with the model between real interactions with the environment.

The step size, the discount factor, and the number of planning updates of the Dyna-SARSA algorithm were initialized as follows:

α	γ	K
0.1	0.9	40

6.3.3 Discussion and results

We first implemented a software version of the complete neurocontroller architecture considering digital hardware constraints. In Section 6.4 we will describe the digital design and some results of a first prototype of the neurocontroller. In the software implementation, we considered 8-bit precision and related problems like overflow, underflow and the limited precision in arithmetic computations. To test the described neurocontroller we used the experimental setup described above in Subsection 6.1.2.

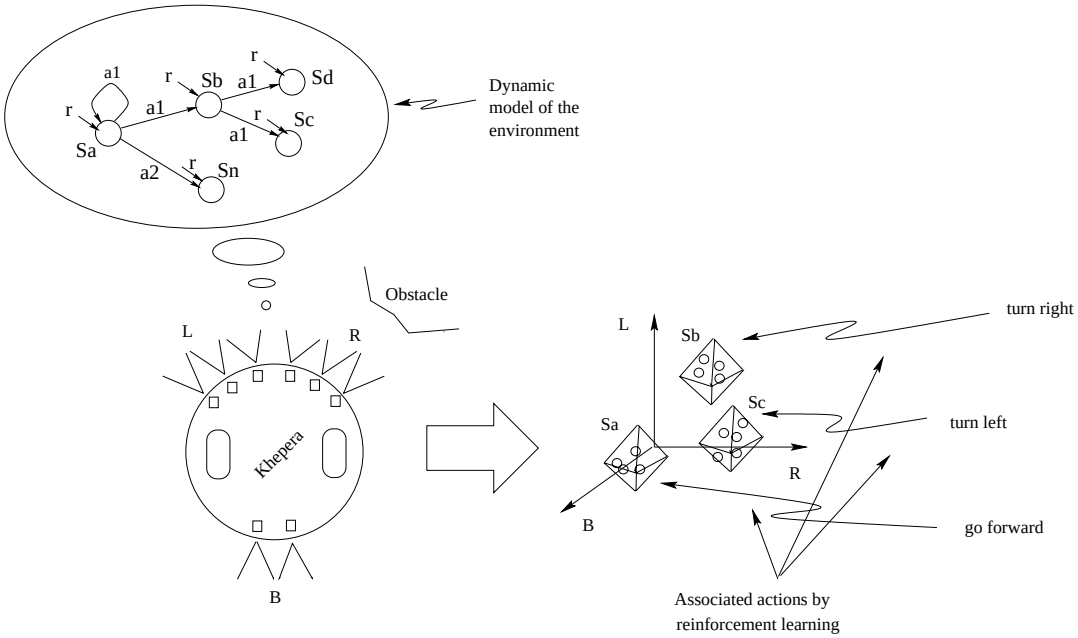


Figure 6.4: Neurocontroller dynamics. The FAST-neurons represent categories (the octahedrons) in the space of sensor activation, which serve as a sparse-coarse-coded function approximator of the input state space. A motor action is then associated to each category by reinforcement learning. The *Dyna*mic model of the environment accelerates learning by a mental rehearsal mechanism.

Following the learning dynamics shown in Figure 6.4, the robot successfully learned an appropriate state-space partition (adaptive categorization) and a perception-to-action mapping, such that the mobile robot was able to avoid obstacles by staying in a quasi-circular path around the obstacle in the center of the arena. Furthermore, the neurocontroller was able to generalize its mappings when we slightly changed the position or the form of the obstacle in the center of the arena.

6.3.3.1 Adaptive categorization

In Section 4.1 we referred to the formation of neuro-ocular pathways in animals as an example of transient neural plasticity. FAST gradually reduces its plasticity as its receptive fields approach the minimum threshold value. Moreover, in our software and hardware implementations we defined a maximum size for the network, which also limits the plastic behavior of the system. As a consequence, after a certain *critical period*, the network's topology does not change overmuch. If some neurons are left, they will be used to allow the system to adapt to possible future dramatic changes in the environment. But, if no neurons are left and we want to allow the system to adapt to changes, a new mechanism should be considered: for example, some kind of counter to indicate if a neuron has not been used for a long time, in which case it might be used to allocate a new category. In Figure 6.5, we show a typical example of the resulting adaptive categorization of FAST in our mobile robot navigation task. The system generally uses the whole set of neurons, which limits its capacity of adaptation to dramatic changes in

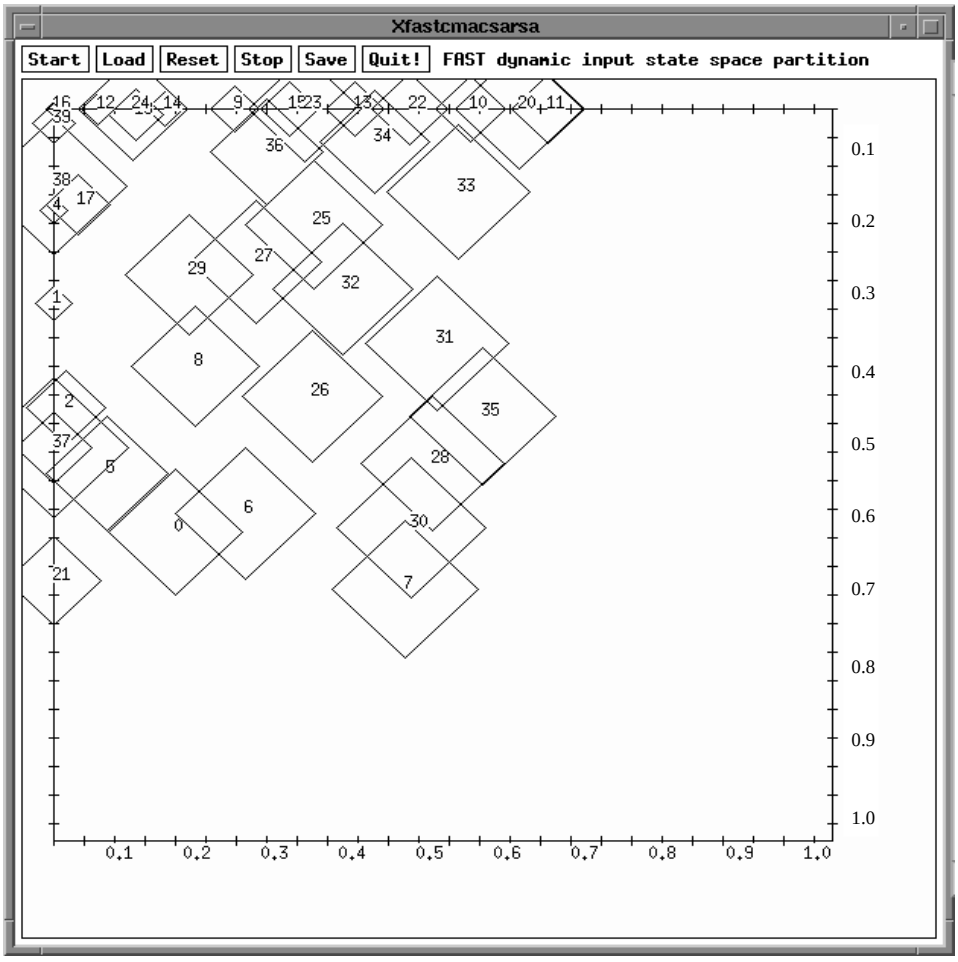


Figure 6.5: Input state space partition. The X and Y axes correspond to the normalized activation of the L and R inputs (eyes) of the system. The rhombi represent the projection of the sensitivity region of the FAST neurons to the XY plane.

the environment. Nevertheless, a reallocating mechanism similar to the one introduced above maybe used, or, some neurons reserved to be used only after a *critical period*. The rhombi of Figure 6.5 represent the projection of the sensitivity region of the 40 FAST neurons on the XY plane. Since our system learns to move forward, and the *go backward* action is only used as part of a pre-wired reflex, such a projection allows us to understand the FAST-learned state space partition. Neuron 21, for example, categorizes the sensor readings that activate the R-input by 60% to 70%, and the L-input by less than 10%. In other words, neuron 21 is activated when the robot senses a very close obstacle to the right, and almost nothing to the left. Similarly, neuron 7 will be activated when the robot senses obstacles to the right and to the left, and neurons 16 and neuron 39 when no obstacles are sensed. The rest of the neurons allow the robot, for example, to start avoiding an obstacle by changing its orientation towards a particular direction.

In Figure 6.6, we show four different state space partitions resulting from a FAST dynamic categorization. The differences in the learned categories are due on the one hand to the environmentally-guided neural circuit building, and on the other hand to the

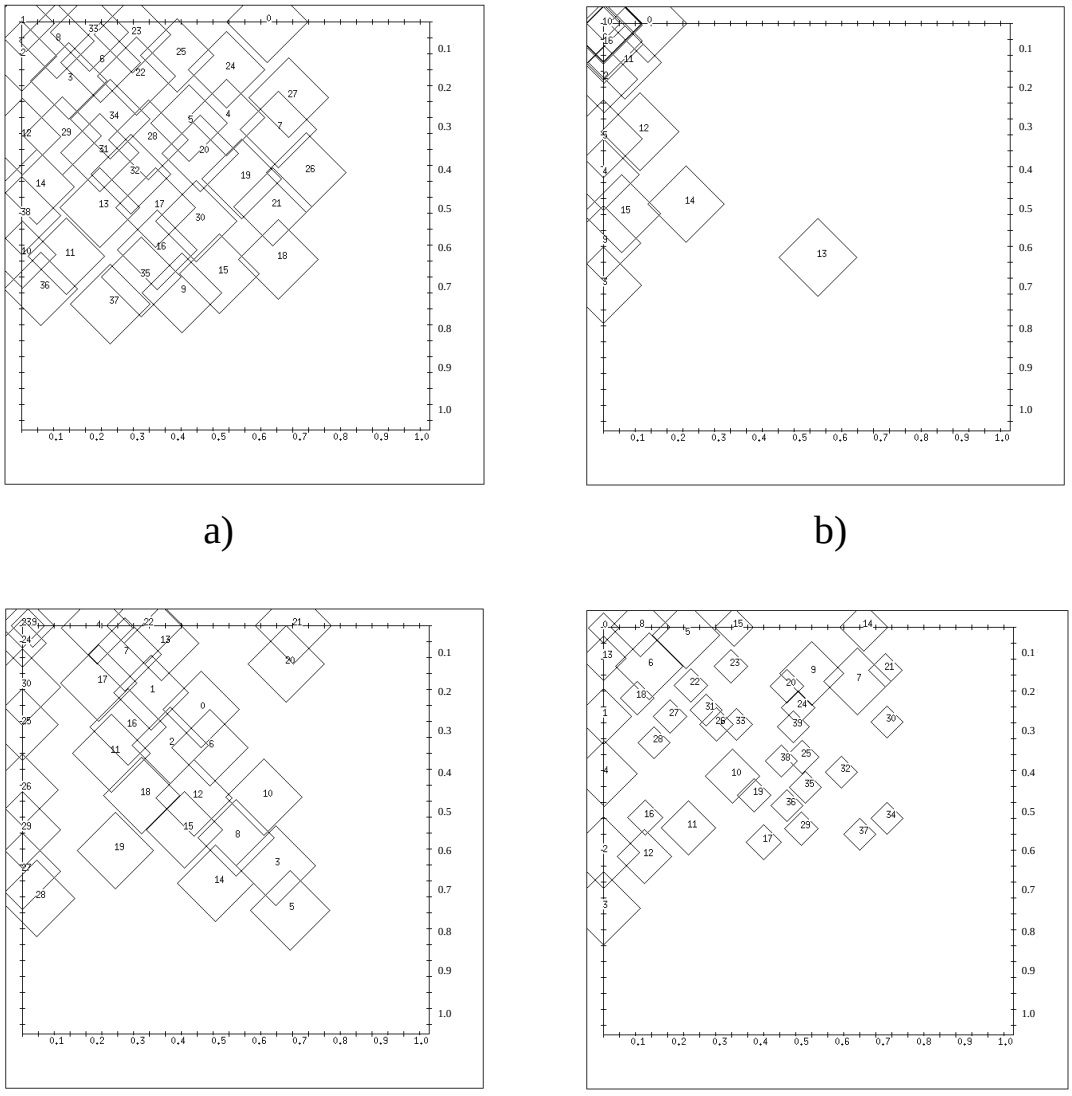


Figure 6.6: Examples of FAST input space partition. The X and Y axes correspond to the normalized activation of the L and R inputs (eyes) of the system. The rhombi represent the projection of the sensitivity region of the FAST neurons to the XY plane.

probabilistic nature of the learning process. The overall behavior of our system resembles the adaptation process observed by Hubel and Wiesel in newborn cats (Section 4.1). If one of the sensor inputs of our robot is not sufficiently activated during its first moments of "life", all neural resources will be used to categorize the activations of the other sensors. As an example, in Figure 6.6a, we present a very homogeneous input space partition: the FAST receptive fields overlap and cover almost all the input space, not unlike a CMAC coarse-coding with adaptive encoding [54]. We have thus called this result a *FAST-CMAC partition*. In Figure 6.6b, we present the resulting FAST categorization of a robot whose L-input sensors where not sufficiently activated during learning, and as a consequence, will rely only on one neuron (neuron 13) to sense obstacles to the left. In Figure 6.6c, we present the case of a robot that started to learn the input space partition in a corner of the workspace. About half of the neurons are situated in a diagonal of the XY plane, and are consequently activated when obstacles are simultaneously present to the left and to the right. The rest of the neurons compensate the other cases of obstacle detection. Finally, in Figure 6.6d, we present an input space partition with very different sizes in the receptive fields of the neurons. This is due to the differences in the frequency of activation of the neurons: while neurons 0 to 9 were initially activated during the early stages of learning and thus have a large threshold (with the exception of neuron 0, which has been repeatedly selected and thus has a smaller receptive field), neurons 16 to 39 where activated later in the learning process and their thresholds were initialized to a small value.

The stability-plasticity dilemma, present in the adaptive categorization process, is crucial in tasks such as our mobile robotics application: for example, if the learned sensor reading categories continue to change indefinitely, it will be very difficult to anticipate future events, since the system will not know when it is in a situation similar to one experimented in the past. In other words, the system does not develop a stable state representation which renders very difficult the estimation of rewards associated with the system's actions.

6.3.3.2 Learning by interaction

In our tests with the Khepera mobile robot, we found Dyna a very useful mechanism to enhance reinforcement learning techniques like SARSA and as useful as learning with eligibility traces (i.e., SARSA(λ)) to handle the problem of delayed reward: the robot was able to learn a state-action mapping that enabled it to move in a quasi-circular path around the obstacle in the center of the workspace.

In Figure 6.7, we show a typical state-action value function, that is, a table of $Q(s, a)$-values. There are 40 states and 3 possible actions: action 0 is 'go forward', action 1 is 'go right', and action 2 is 'go left'. The height of the bars represent the Q-values, which take values in the range [-1 0]. In this figure, the smaller bars represent the actions with larger reward estimates, that is, the actions that are more likely to be taken from such state. As an example, consider state 0 in Figure 6.7. The action that will be most probably taken from such state is action 0, or 'go forward'. In state 14, the system will turn right or turn left, but will try to avoid going forward.

We initialized the Q-table to zeros everywhere. As in our previous experiences with

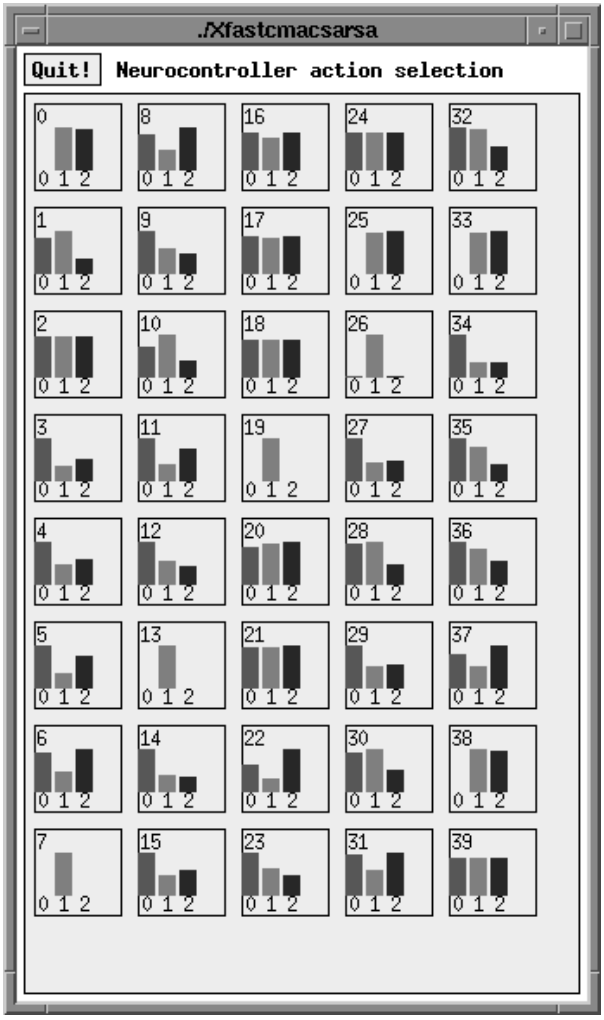
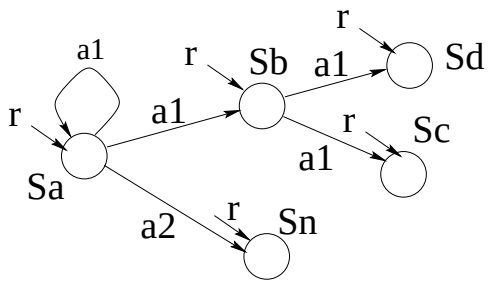


Figure 6.7: State-action function value. The bars represent the Q-values of states 0 to 39 and actions 0 (go forward), action 1 (go right), and action 2 (go left). The smaller the bar, the better the action to take from each state.

maze learning, this values correspond to optimistic values (our system receives only '0' or '-1' as reinforcement), stimulating the exploration of actions during the first stages of learning. We found this technique a useful compensation for the fixed low rate of exploration in our system. Indeed, we used an ϵ-greedy action selection mechanism, with ϵ equal to 0.01. In other words, our system exploits with a probability of 99%, and explores only 1% of the time.

The Dyna model of the world is dynamically constructed using a table that stores all observed next states along with their frequency of occurrence and an average of the received reward (Figure 6.8). Note that the taking of a particular action a_1 from a given state S_a can result in different next states s' (S_a or S_b) due to the dynamic state representation and the non-deterministic nature of the problem.

The dynamic state representation in our system (note that it is determined by the FAST network, and that each new run of the system may associate a particular neuron to different sensor readings) renders very difficult the comparison of various learned



	S_a	S_b	...	S_n	r
S_a a_1	15	5	...	0	-1
S_a a_2	0	80	...	2	0
...	.	.	.	.	.
...	.	.	.	.	.
S_b a_1	0	0	...	0	0
...	.	.	.	.	.
...	.	.	.	.	.

Figure 6.8: Dyna for non-Markovian environments. The Dyna model stores all observed next states along with their frequency of occurrence and an average of the received reward. During the planning updates, the next state of a transition is randomly chosen according to the stored frequencies.

state-to-action mappings. Moreover, a particular FAST-Dyna-SARSA behavior may enable the robot to avoid obstacles without being the best and more robust possible strategy. Indeed, our experiments have not been designed to let the system learn the 'best' obstacle-avoidance behaviors, since the system is only punished when it crashes or turns on itself, and no extra information is provided. Nevertheless, these experiments with a real robot, real sensors and a real workspace allowed us to study both adaptive categorization and reinforcement learning and test our neurocontroller system. The philosophy of these experiments relied on what is predicated by Sutton: *"approximate the solution and not the task"*, that is, even though we have an imperfect representation of the environment, we can nevertheless obtain good results.

6.4 FPGA neurocontroller design

As we mentioned, we have implemented and tested a software version of the complete neurocontroller architecture considering digital hardware constraints. Even though we have not implemented the entire neurocontroller using FPGA circuits, this section is intended to describe the realization of two printed circuit boards to host the neurocontroller, the neurocontroller design, and the implementation results of a 16-neuron FAST network.

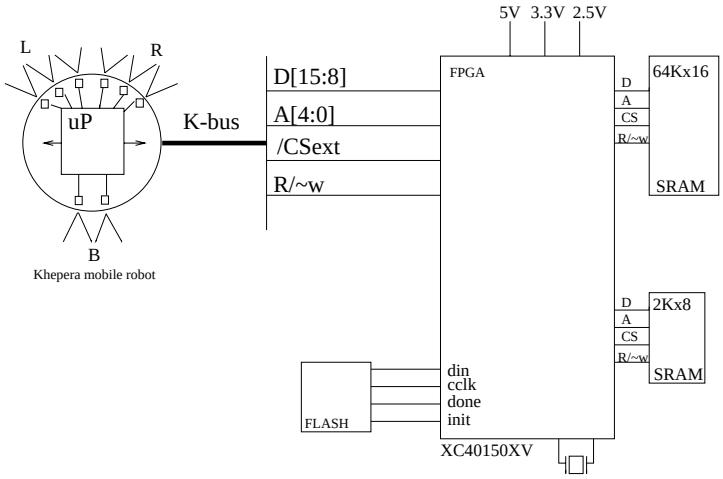


Figure 6.9: Diagram of interconnections between the robot and the components of the neurocontroller.

6.4.1 The printed circuit boards

Two printed circuit boards hosting the neurocontroller architecture have been developed and tested. One was conceived for communication between the Khepera bus extension and the FPGA board, and to generate the 3.3V and 2.5V FPGA power voltage levels from the 5V standard power system of the mobile robot. The second board hosts the FPGA, a configuration FLASH memory, and two SRAM memories connected to the FPGA. The FPGA is a Xilinx XC40150XV-BGA432 [239], containing 5,184 configurable logic blocks and 11,520 flip flops and allowing the implementation of digital circuits with approximately 150,000 equivalent gates. The configuration memory is an AM29F040 FLASH memory of 512Kbytes (4 Mbits), storing the 3,373,448 configuration bits of the FPGA. The other two SRAM chips are an IDT71016 and an IDT6116, storing 1 Mbit (64Kx16-bit) and 16 Kbit (2Kx8-bit) of information respectively. The 1 Mbit memory is intended for implementing the *Dyna* model of world (Section 5.6), while the other SRAM can be used for future extensions of the neurocontroller. In particular, we have thought of using such memory for the implementation of a Dyna-queue algorithm [151] (an extension of the Dyna algorithm) or for storing values to be shared between two or more configurations of the FPGA. Figure 6.9 shows the interconnections between the robot and the components of the neurocontroller, and Figure 6.10 shows a photo of the robot with the neurocontroller's printed circuit board.

6.4.2 Neurocontroller design

We have designed a 16-neuron FAST-Dyna-SARSA system. The 16-neuron FAST system is an extension of the 3D-input FAST network described in Subsection 4.5.4. The implementation of the new network used 85% of the configurable logic blocks (CLBs) of the Xilinx XC40150XV chip. The maximal pin-to-pin delay was approximately 90 ns and the minimum period was approximately 175 ns using a XC40150XV-9 device, thus enabling a maximum clock frequency of about 5.5 MHz (for a particular run of the par-

Figure 6.10: Photo of the Khepera mobile robot and our Xilinx XC40150XV FPGA-based board. [Photo by André Badertscher]

tition, placement, and routing process). As in the case of the implementation described in Chapter 4, a maximum of 66 clock cycles is needed for a learning step, thus enabling the introduction of a new input vector approximately every 12 μs.

The reinforcement learning (Dyna-SARSA) module is a simplified version of the AHC implementation presented in Chapter 5. The system stores only one state-action value $Q(s, a)$ instead of separated values for the *actor* and the *critic*, thus simplifying the connectivity and the sequencer of the system. The system uses an ϵ-greedy action selection instead of the softmax-like mechanism used in the AHC implementation, sidestepping the need for a sigmoid function. Finally, since the state-action values are constrained to the range [-1 0], there is no need for normalization operation as in the previous implementation. However, the new system will include a small amount of additional hardware to implement the update of the *dl* and *dr* values described in Subsection 6.1.2, the control of the planning updates, and the storing of the Dyna model.

The planning updates can be implemented very easily, a memory to store the Dyna model, and some extra states in the sequencer to control the use of the model to generate offline state transitions, which will allow the system to update the state-action values offline.

To test this design, we have realized new runs with a 16-neuron FAST-Dyna-SARSA software implementation of the neurocontroller (instead of the 40-neuron system described above). The system is again able to learn a quasi-circular path around the obstacle in the center of the arena. Figure 6.11 shows the experimental setup used for these tests. Therefore, a last implementation was conceived such that the FPGA-based

Figure 6.11: Obstacle avoidance task. [Photo by André Badertscher]

board implements the 16-neuron FAST network, and the 68331 microcontroller in the Khepera robot implements the Dyna-SARSA learning algorithm. The cable is only used to bring power to the system and to visualize the behavior of the system for debugging.

6.5 Neurocontroller architecture with short-term memories

In Section 5.7, we described the problem of having partial information about the current state of the environment in maze tasks. Similarly, in mobile robotics, the agent does not have access to its global position in the workspace, and has to rely on its own sensors to properly behave. Moreover, mobile robots have noisy and non-linear sensors and suffer from the problem of *hidden-state* [116] or *perceptual aliasing* [227] which renders the problem of navigation a non-Markovian task.

The problem of partial observability and value function approximation are closely related: even though they are not identical, to a first approximation they seem to require the same solutions [199]. In our experiments, most issues related to this problem were solved by neural network function approximation. However, in more complex workspaces, the system might experience the same sensory readings while in different situations. For simplicity, the maze task shown in figure 6.12 illustrates this problem.

We have developed an extension of the function approximation and incremental construction of the state representation in our system: the idea is to introduce an expansion of the state representation over space and time, by means of short-term memories to disambiguate identical sensory inputs observed in different states (Figure 6.13).

A new FAST-neuron is added whenever the oscillation of the Q-values (i.e., the expected reward) of a given FAST-category is higher that a certain threshold. In fact,

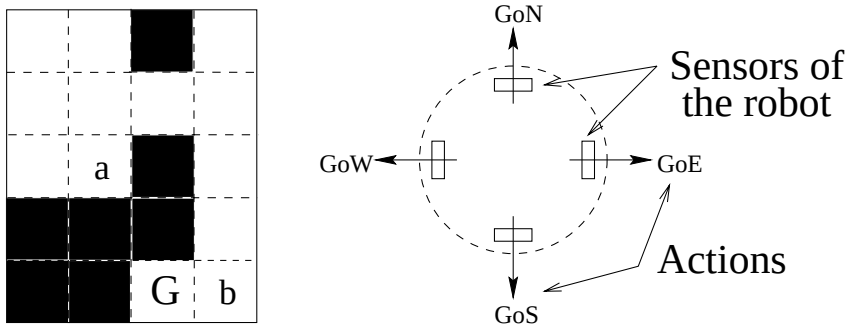


Figure 6.12: Maze-like partially observable environment: the sole information from the sensors of the robot is not enough to determine the optimal actions in the given maze. In the cell positions 'a' and 'b' of the maze, an abstract 4-sensor robot senses obstacles to the right and to the south, though the corresponding optimal actions are different: 'GoNorth' and 'GoWest'.

Q-learning is still able to solve a partially observable problem, but it is not capable of finding the optimal solutions. In the example of Figure 6.12, the learned Q-values for states 'a' and 'b' will oscillate because there is not a unique optimal action for both cell positions. Therefore, by extending the state representation over time, it will be possible to assign different FAST-categories to the cell positions 'a' and 'b' and associate different Q-values such that the system can select different actions.

In order to differentiate cell positions 'a' and 'b' we compute two values 'A' and 'B' as in [169]:

$$A(s,a) = A(s,a) + \sigma \Delta Q(t) + (1-\sigma)A(s,a) \ , \tag{6.3}$$

$$B(s,a) = B(s,a) + \sigma |\Delta Q(t)| + (1-\sigma)B(s,a) \ , \tag{6.4}$$

where σ is a constant value. After some iterations, if the Q-values oscillate (because there is not a unique optimal action), the value $A(s,a)$ gets close to zero (since it considers the ΔQ sign), though the value $B(s,a)$ continue to increase (since it always adds something positive to the previous $B(s,a)$ value). Therefore, if $B(s,a)$ exceeds a given threshold, two cell positions with equal environment observations but different optimal actions can be disambiguated.

Finally, a new state can be added to the learned reactive system as a function of the current environment's observation and some additional information (for example, the previous state of the environment). This is equivalent to introducing a short-term memory in the FAST-neurons, so that one of these new FAST-neurons is activated only if the pattern of activation of the sensors is sufficiently similar to the pattern stored in its corresponding long-term memory (i.e., the reference vector), and if the temporal pattern of activation of neurons matches a certain sequence of states. This is what we have called *spatiotemporal FAST clustering* in Chapter 4.

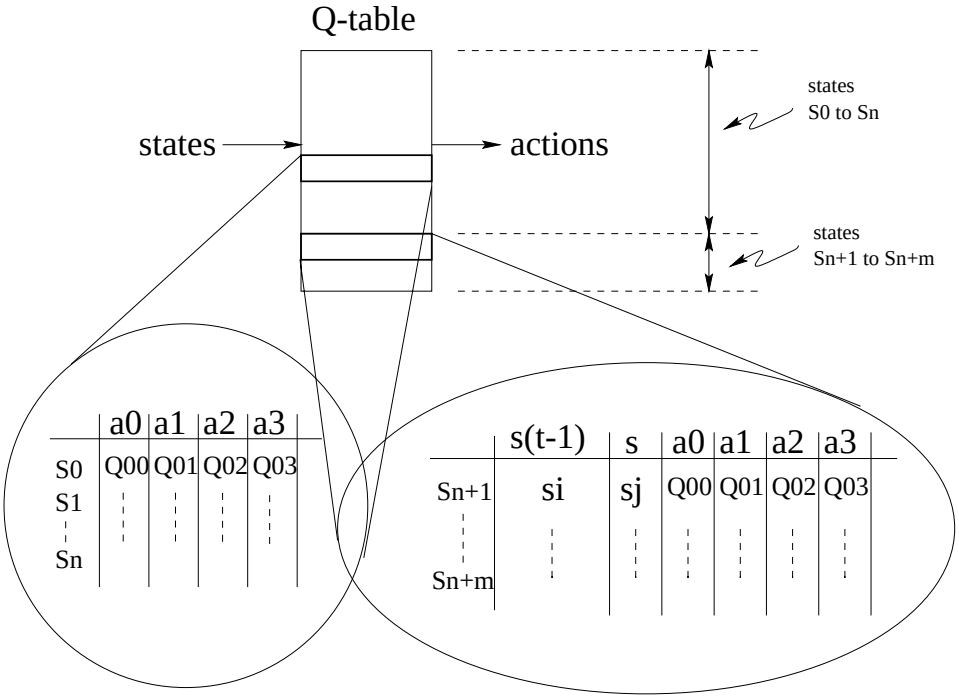


Figure 6.13: Expansion of the state representation over time: the states S_0 to S_n correspond to FAST-categories in the sensor activation space of the system; states S_{n+1} to S_{n+m} correspond to FAST-categories in an extended temporal sensor activation space. One of the latter FAST-neurons (or categories) is activated only if the pattern of activation of the sensors is sufficiently similar to the stored pattern in such neuron, and the temporal pattern of activation of neurons matches a certain sequence of states. As an example, S_{n+1} is activated if the previous state was S_i and the current pattern of activation of the sensors corresponds to state s_j.

6.6 Summary

We have presented an integrated digital neurocontroller architecture for mobile autonomous robots. Our architecture was motivated by the possibility of digital implementation using programmable devices, and provides a learning robot with the capability to autonomously categorize input data from the environment, to deal with the stability-plasticity dilemma, and to learn a state-to-action mapping that enables it to navigate in a workspace while avoiding obstacles. The neurocontroller architecture is composed of two main modules: an adaptive categorization module, implemented by the FAST neural architecture, and a model-based reinforcement learning module (Dyna-SARSA). Reinforcement learning algorithms have been shown to be an interesting paradigm for learning (autonomously) by interaction. Constructive artificial neural networks like FAST provided to be an interesting approach to attempt to solve several learning related problems: automatic partition of the input state space, approximation and generalization, partial observability of the environment due to noisy and non-linear sensors, and neurocontroller structure adaptation. Simulated experience, or mental rehearsal (using a *Dyna*mic incremental model of the environment) was successfully used to accelerate convergence of the value functions in reinforcement learning. The resulting structure-adaptable neurocon-

troller was used with an autonomous mobile robot to learn to avoid obstacles by staying in a quasi-circular path around an obstacle in the center of an arena. The neurocontroller was able to generalize its mappings when we slightly altered the position or the form of the obstacle in the center of the arena. The learning dynamics of the neurocontroller were described in detail. Two printed circuit boards were designed to host an FPGA-based neurocontroller implementation. The design and some implementation results of a first prototype are described. Finally, an extension of the function approximation and incremental construction of the state representation was introduced to attempt to solve the problem of partial observability of the environment in real-world workspaces.

Chapter 7

Conclusions

> "The most incomprehensible thing about the universe
> is that it is comprehensible."
>
> -A. Einstein

Engineers have been trying for a long time to develop bio-inspired computing machines by drawing their inspiration from nature. In particular, researchers at the Logic Systems Laboratory have used field-programmable logic devices (FPGAs) to implement bio-inspired hardware systems, capable of evolution, growth, and self-repair [130], along all the three axes of the POE model [193]. The focus of this thesis is the epigenetic axis, our goal being the development of digital artificial neural networks with online learning capabilities. We argue that true learning systems adapt to problems not only by changing the strength of the interconnections between their computational elements (as is the case in most neural network models) but also by dynamically reconfiguring their structure, an hypothesis which gave rise to the idea of developing structure-adaptable digital neural networks using field-programmable logic devices.

Our goal was met in most respects. We have developed an artificial neural network system which is well suited for digital implementation, called FAST for Flexible Adaptable-Size Topology. This neural-based system implements an adaptation of learning models inspired from neurophysiology and statistical clustering techniques. While FAST is not the first neural network based on structure adaptation, it is special in that it does not require intensive computation to learn and adapt its structure, and can thus exist as an stand-alone learning machine that can be used, for example, in a real-time control application. FAST has been conceived to be implemented using programmable logic devices. However, the reconfigurability of such devices has not yet been exploited. The development of such a system remains a future research goal.

As far as online learning is concerned, the FAST learning algorithm was developed to handle the problem of dynamic categorization or online clustering: the system receives as input a stream of sensations coming from its environment and is asked to construct a model to explain the observed properties of the input patterns. No external supervisor provides the desired outputs or rewards. The model must reflect the particular statistical structure of the overall collection of input patterns. Nevertheless, a different learning

process had to be considered to allow a system to generate behavioral responses as a function of its sensations. Indeed, other types of learning, such as reinforcement learning, seem to govern spatial and motor skill acquisition. We have thus combined the capabilities of the FAST neural architecture with reinforcement learning techniques to develop a neurocontroller architecture. In particular, we have coupled FAST with an adaptive heuristic critic (AHC) network to implement a neurocontroller to solve the inverted pendulum problem, and with Dyna-SARSA learning to control an autonomous mobile robot. Moreover, we have devised a digital implementation of these neurocontrollers and developed an FPGA board to control the Khepera mobile robot in a navigation-learning task.

As far as structure adaptation is concerned, the resulting neurocontroller architecture does indeed adapt its topology, if only to a certain extent: the FAST algorithm entails a change in the size of the network (i.e., the number of active neurons), but does not allow more complex topology adaptations. However, it should be noted that a change in the number of active neurons in FAST implies further changes in the topology of a FAST-based neurocontroller. For example, in the case of the FAST-AHC neurocontroller, the activation of a new FAST neuron implies the activation of two new synapses to the actor and critic neuron-like elements composing the AHC system. Similarly, if a FAST neuron is deactivated, its two synapses together with the actor and critic neuron-like elements, will also be deactivated, and remain available for future use. This process resembles the neural plasticity phenomenon observed in newborn cats: the neuronal connections in the kittens' eyes reorganize when one eye is artificially sutured and does not receive normal activations from its environment.

7.1 Original contributions

The domain of bio-inspired systems is the result of the convergence of ideas from very diverse fields. While my contributions cannot be classified as a complete new domain of research, this thesis is as an example of a multidisciplinary endeavor. My most original contribution is probably the convergence of ideas from unsupervised learning, reinforcement learning, reconfigurable hardware, and autonomous robotics. In this section, I will describe my original contributions to the domain of bio-inspired systems within each of the main chapters.

Chapter 1 introduces the POE model of bio-inspired hardware systems, which helps define the focus of this thesis along the epigenetic axis of inspiration. My research on learning hardware systems certainly helped the definition of the POE model. Moreover, I have contributed with the definition of learned and learning systems.

Chapter 2 and Chapter 3 are intended to provide some background on artificial neural networks and neural hardware. My main contribution in these chapters is to point out the analogy between hardware reconfigurability and neural plasticity, and the fact that programmable logic devices, besides rapid prototyping and fast operation, enable us to implement artificial neural systems with modifiable topologies.

Chapter 4 contains the description of the FAST neural architecture, starting with the description of some important aspects drawn from biology, and concluding with a digital

implementation of the system and its use for image processing. My main contribution in this chapter is the adaptation of existing structure-adaptable neural networks to develop the learning algorithm implemented by the FAST network. In particular, we considered the digital constraints imposed by reconfigurable hardware to develop a computationally non-intensive learning algorithm, studied the consequences of not using a winner-take-all operator within the dynamic clustering process, and developed a probabilistic pruning mechanism to compensate this lack. Another contribution is the implementation of FAST using programmable logic devices. This was, to our knowledge, the first FPGA implementation of an on-chip learning artificial neural model with modifiable topology. Finally, we pointed out the basics of a spatiotemporal clustering mechanism, based on the FAST premises.

Chapter 5 is concerned with reinforcement learning techniques. My first contribution in this chapter is dealing with the problem of using modern reinforcement learning techniques to learn to play the game of Blackjack and analyzing the learned strategies. My second original contribution is to analyze the dynamics of reinforcement learning with eligibility traces and model-based reinforcement learning from the computational point of view, and to perform a comparison of such techniques in Markovian and non-Markovian tasks. The result of such comparison is the development of the neurocontroller architecture described in Chapter 6. My final contribution in this chapter is the design, hardware realization, and test of a neurocontroller implementing a FAST and an adaptive heuristic critic (AHC) architecture.

Chapter 6 presents a neurocontroller architecture for autonomous mobile robot navigation. My main contribution is the synthesis of all previous ideas in the development of a modular architecture implementing a FAST and a model-based reinforcement learning algorithm (Dyna-SARSA). Another original contribution is the design of an FPGA-board to host the neurocontroller architecture. I was particularly interested in the idea of providing a hardware coprocessor that allows a system to realize, at electronic speeds, a sort of a 'mental rehearsal' process to ameliorate its behavior.

7.2 Challenges and further research directions

There are many interesting issues that are crucial for further development of the epigenetic axis of bio-inspired hardware systems, and particularly of autonomous learning systems. In this section I will try to point out some of the most important issues related to this thesis.

7.2.1 Extending FAST

FAST has been shown to be a fast learning alternative for pattern clustering tasks. However, to allow the development of adequate clustering in complicated probability density distributions of the data, it would be interesting to store a set of two different thresholds per unit, so that the shape of the sensitivity regions would be closer to the ellipsoids of statistical clustering techniques. The preliminary results on neurocontroller structure adaptation are very encouraging and suggest a large number of future experiments.

A potential application of FAST, not yet fully explored, is the ability to adapt to quick changes in the environment. Millan [137] found poor generalization capabilities in a similar architecture based on GAL (the supervised version of GAR, the Grow and Represent algorithm described in Chapter 4) when the system was tested on environments different from those previously experienced. However, we believe the FAST architecture holds greater potential by offering a fast and plastic learning platform for non-stationary environment applications. A promising alternative to improve our system is to incorporate the reinforcement signals into the FAST network's dynamics. This idea was inspired by Carpenter and Grossberg who suggested "placing the vigilance parameter under external control, to be increased, for example, when the network is 'punished' for failing to distinguish two inputs that give rise to different consequences" [37]. Indeed, I have thought of a nice experiment that can be used to test the *continual learning* capabilities of our architecture by using several environments with increasing complexity in its shape and obstacles. Such an experiment would be similar to the abstract maze experiences of Ring [169], who tested a high-order constructive learning algorithm with maze tasks of increasing complexity. The system was able to sequentially learn those mazes, and required a smaller computational effort to solve the more complex maze than solving it from scratch.

7.2.2 Hardware reconfigurability

Semiconductor technology is changing very fast. New devices like the FIPSOC chip [145], which contains a microcontroller, some digital reconfigurable logic, and some programmable analog circuits, should enable the development of hardware-oriented learning algorithms (HOLA). Indeed, a big challenge for future research is the development of a learning algorithm sufficiently simple to be implemented using reconfigurable hardware devices but powerful enough to reconfigure the complete pattern of interconnection between very simple processing elements, implemented in the same chip. This would be similar to implementing the idea of *unorganized* learning machines defined by Turing in the late 1940s [213].

A possible way to achieve this is by developing computational non-intensive evolutionary learning algorithms based on the so-called *classifier systems*, developed by Holland [86], and on its more recent versions, called ZCS and XCS [233, 234]. Such systems categorize patterns of sensory activation by using a very simple alphabet composed of 0's, 1's, and don't-care's (#), very similar to the weightless models of Aleksander [5].

7.2.3 Bio-inspired hardware dualism

To conclude with a more "philosophical" issue, I have observed a kind of dualism in our current bio-inspired learning models: from a conceptual standpoint, there is a network topology and a learning algorithm. In other words, there are always a predetermined structure composed of interconnected processing units and a procedure to adapt (hopefully) both the dynamics of the processing elements and the way they are interconnected. This dualism is analogous to hardware and software in computing machines, or to data paths and control paths in hardware systems.

This duality is understandable since computing machines have so far been implemented as *programmable* machines. However, the *learning paradigm* should enable us to think of a new kind of machines, where, effectively, learning by examples and interaction replace programming (without needing to emulate such principles using a programmable computing machine). In this kind of machines, the learning algorithm would emerge from the dynamics of the interconnection of the processing elements, which may be the key to realize a system endowed with "semantics" (i.e., a system that is capable of associating a meaning to the symbols it uses for computing) [186, 187], and not merely with "syntax", as it is the case of our current computing machines.

Bibliography

[1] J.S. Albus. A New Approach to Manipulator Control: The Cerebellar Model Articulation Controller (CMAC). *Journal of Dynamic Systems, Measurement and Control*, 97:220–227, September 1975.

[2] I. Aleksander. Artificial Neuroconsciousness: An Update. In J. Mira and F. Sandoval, editors, *From Natural to Artificial Neural Computation*, volume 930 of *Lecture Notes in Computer Science*, pages 566–583. Springer, 1995.

[3] I. Aleksander. *Impossible Minds. My Neurons My Consciousness.* Imperial College Press, 1996.

[4] I. Aleksander and E.H. Mamdani. Microcircuit Learning Nets: Improved Recognition by Means of Pattern Feedback. *Electronic Letters*, 4(20):425–426, October 1968.

[5] I. Aleksander and H. Morton. *An Introduction to Neural Computing.* International Thomson Computer Press, second edition, October 1995.

[6] A. E. Alpaydin. *Neural Models of Incremental Supervised and Unsupervised Learning.* PhD thesis, Swiss Federal Institute of Technology-Lausanne, Lausanne, EPFL, 1990. Thesis 863.

[7] J. Alspector, J. Gannett, S. Haber, M. Parker, and R. Chu. A VLSI-Efficient Technique for Generating Multiple Uncorrelated Noise Sources and Its Application to Stochastic Neural Networks. *IEEE Transactions on Circuits and Systems*, 38(1):109–123, January 1991.

[8] Altera. *Data Book.* Altera Corporation, Santa Clara, 1996.

[9] C.W. Anderson. Strategy Learning with Multilayer Connectionist Representations. In *Proceedings of the Fourth International Workshop on Machine Learning*, pages 103–114, Irvine, CA, 1987.

[10] C.W. Anderson. Q-Learning with Hidden-Unit Restarting. In *Advances in Neural Information Processing Systems*, pages 81–88. Morgan Kaufmann Publishers, 1993.

[11] D. Andre, N. Friedman, and R. Parr. Generalized Prioritized Sweeping. In M.I. Jordan, M.J. Kearns, and S.A. Solla, editors, *Advances in Neural Information Processing Systems*, volume 10. The MIT Press, 1998.

[12] C. Aoki and P. Siekevitz. Plasticity in Brain Development. *Scientific American*, 259:34–42, December 1988.

[13] M.A. Arbib. Applications and Implementations. In M.A. Arbib, editor, *Handbook of Brain Theory and Neural Networks*, chapter Road Maps, pages 41–44. MIT Press, 1995.

[14] R.C. Arkin. Reactive Robotic Systems. In M.A. Arbib, editor, *Handbook of Brain Theory and Neural Networks*, pages 793–796. MIT Press, 1995.

[15] R.C. Arkin. *Behavior-Based Robotics*. The MIT Press, Cambridge, MA, 1998.

[16] T. Ash and G. Cottrell. Topology-Modifying Neural Network Algorithms. In M.A. Arbib, editor, *Handbook of Brain Theory and Neural Networks*, pages 990–993. MIT Press, 1995.

[17] J. Austin, editor. *RAM-Based Neural Networks*, volume 9 of *Progress in Neural Processing*. World Scientific, February 1998.

[18] A. Avizienis. Signed-digit number representations for fast parallel arithmetic. In *IRE Transactions on elecronic computers*, volume 10, pages 389–400, 1961.

[19] A. Baddeley. Memory. In *The MIT Encyclopedia of the Cognitive Sciences*, pages 514–517. The MIT Press, 1999.

[20] S.L. Bade and B.L. Hutchings. FPGA-Based Stochastic Neural Networks Implementation. In D.A. Buell and K.L. Pocek, editors, *IEEE Workshop on FPGAs for Custom Computing Machines*, pages 189–198, Los Alamitos, CA, April 1994. IEEE Computer Society Press.

[21] K. Balakrishnan. Evolutionary Design of Neural Architectures - A Preliminary Taxonomy and Guide to Literature. Technical Report CS-TR-95-01, Department of Computer Science, Iowa State University, Ames, USA, January 1995.

[22] A.G. Barto. Reinforcement learning in motor control. In M.A. Arbib, editor, *Handbook of Brain Theory and Neural Networks*, pages 809–812. MIT Press, 1995.

[23] A.G. Barto, R.S. Sutton, and C.W. Anderson. Neuronlike Adaptive Elements That Can Solve Difficult Learning Control Problems. *IEEE Transactions on Systems, Man and Cybernetics*, 13(5):834–846, 1983.

[24] R. Bellman. *Dynamic Programming*. Princeton University Press, 1957.

[25] R. Bellman. *Adaptive Control Processes: A Guided Tour*. Princeton University Press, 1961.

[26] D. Benitez-Diaz and J. Garcma-Quesada. Learning Algorithm with Gaussian Membership Function for Fuzzy RBF Neural Networks. In *From Natural to Artificial Neural Computation*, pages 527–534, Springer Verlag, 1995.

[27] H. Berenji and P. Khedkar. Learning and Tuning Fuzzy Logic Controllers through Reinforcements. In *IEEE Transactions on Neural Networks*, pages 724–740, September 1992.

[28] J.-L. Beuchat. Réalisation de réseaux neuronaux en logique stochastique. Technical report, Swiss Federal Institute of Technology-Lausanne, 1996. (Semester project report in French).

[29] J.A. Boyan. Modular neural networks for learning context-dependent game strategies. Master's thesis, University of Cambridge, Department of Engineering and Computer Laboatory, 1992.

[30] R.A. Brooks. Intelligence Without Reason. Technical Report A.I. Memo 1293, Massachusetts Institute of Technology, April 1991.

[31] R.A. Brooks. New Approaches to Robotics. *Science*, 253:1227–1232, 13 September 1991.

[32] J. Bruske, I. Ahrns, and G. Sommer. An integrated architecture for learning of reactive behaviors based on dynamic cell structures. *Robotic and Autonomous Systems*, 22:87–101, 1997.

[33] L.I. Burke. Clustering Characterization of Adaptive Resonance. *Neural Networks*, 4:485–491, 1991.

[34] G.A. Carpenter. Distributed ART networks for learning, recognition, and prediction. In *World Congress on Neural Networks*, pages 333–344, San Diego, CA., September 15-18 1996. International Neural Network Society.

[35] G.A. Carpenter and S. Grossberg. ART2: Stable Self-organization of Pattern Recognition Codes for Analog Input Patterns. *Applied Optics*, 26(21):4919–4930, November 1987.

[36] G.A. Carpenter and S. Grossberg. A massively parallel architecture for a self-organizaing neural pattern recognition machine. *Comput. Vis., Graph., Image Proc.*, 37:54–115, 1987.

[37] G.A. Carpenter and S. Grossberg. The ART of Adaptive Pattern Recognition by a self-organizing neural network. *IEEE Computer*, pages 77–88, March 1988.

[38] G.A. Carpenter and S. Grossberg. ART3: Hierarchical search using chemical transmitters in self-organizing pattern architectures. *Neural Networks*, 3:129–152, 1990.

[39] G.A. Carpenter and S. Grossberg. Adaptive resonance theory (ART). In M.A. Arbib, editor, *Handbook of Brain Theory and Neural Networks*, pages 79–82. MIT Press, 1995.

[40] G.A. Carpenter, S. Grossberg, N. Markuzon, J.H. Reynolds, and D.B. Rosen. Fuzzy ARTMAP: A neural network architecture for incremental supervised learning of analog multidimensional maps. *IEEE Transactions on Neural Networks*, 3:698–713, 1992.

[41] G.A. Carpenter, S. Grossberg, and J.H. Reynolds. Supervised real-time learning and classification of nonstationary data by a self-organizing neural network. *Neural Networks*, 4:565–588, 1991.

[42] G.A. Carpenter, S. Grossberg, and D.B. Rosen. Fuzzy ART: Fast stable learning and categorization of analog patterns by an adaptive resonance system. *Neural Networks*, 4:759–771, 1991.

[43] J.-P. Changeux. *L'homme neuronal*. Collection Pluriel. Fayard, 1985.

[44] J.-P. Changeux and A. Danchin. Selective stabilisation of developing synapses as a mechanism for the specification of neuronal networks. *Nature*, 264:705–712, December 1976.

[45] G. Chechik and I. Meilijson. Synaptic Pruning in Development: A Novel Account in Neural Terms. *Neural Computation*, 10(7):1759–1777, October 1998.

[46] J. Cloutier and P.Y. Simard. Hardware implementation of the backpropagation without multiplication. In *Proceedings of the Fourth International Conference on Microelectronics for Neural Networks and Fuzzy Systems Microneuro'96*, pages 46–55. IEEE, September 1994.

[47] C. Cox and E. Blanz. GANGLION: A Fast Field Programmable Gate Array implementation of a Connectionist Classifier. *IEEE Journal of Solid State Circuits*, 27(3):288–299, March 1992.

[48] C. Darwin. *The Origin of Species by Means of Natural Selection*. John Murray, London, 1859.

[49] G. de Trémiolles, P. Tannhof, B. Plougonven, C. Demarigny, and K. Madani. Visual probe mark inspection, using hardware implementation of artificial neural networks in VLSI production. In R. Moreno-Díaz J. Mira and J. Cabestany, editors, *Biological and Artificial Computation: From Neuroscience to technology*, volume 1240, pages 1374–1383, Lecture Notes in Computer Science, 1997. Springer Verlag.

[50] S. Dehaene and J.-P. Changeux. Development of Elementary Numerical Abilites: A Neuronal Model. *Journal of Cognitive Neuroscience*, 5(4):390–407, 1993.

[51] T. Dietterich. Machine learning. In *The MIT Encyclopedia of the Cognitive Sciences*, pages 497–498. The MIT Press, 1999.

[52] M. Dorigo. Editorial Introduction to the Special Issue on Learning Autonomous Robots. In *IEEE Transactions on Systems, Man and Cybernetics*, pages 361–364, June 1996.

[53] R. Douglas and M. Mahowald. Silicon Neurons. In M.A. Arbib, editor, *Handbook of Brain Theory and Neural Networks*, pages 871–875. MIT Press, 1995.

[54] M. Eldracher, A. Staller, and R. Pompl. Adaptive Encoding Strongly Improves Function Approximation with CMAC. *Neural Computation*, 9:403–417, 1997.

[55] J.G. Eldredge and B.L. Hutchings. Density Enhancement of a Neural Network Using FPGAs and Run-Time Reconfiguration. In D.A. Buell and K.L. Pocek, editors, *IEEE Workshop on FPGAs for Custom Computing Machines*, pages 180–188, Los Alamitos, CA, April 1994. IEEE Computer Society Press.

[56] M. Ercegovac. On-line arithmetic: an overview. In *SPIE, Real Time Signal Processing VII*, pages 86–93, 1984.

[57] S.E. Fahlman and C. Lebiere. The Cascade-Correlation Learning Architecture. In *Advances in Neural Information Processing Systems*, pages 524–532. Morgan Kaufmann Publishers, 1990.

[58] M.C. Fairhurst and I. Aleksander. Natural Pattern Clustering in Digital Learning Nets. *Electronic Letters*, 7(24):724–726, October 1971.

[59] E. Fiesler. Comparative Bibliography of Ontogenic Neural Networks. *Proceedings of the International Conference on Artificial Neural Networks (ICANN'94)*, 1994.

[60] G. Fischback. Mind and Brain. *Scientific American*, pages 24–33, September 1992.

[61] D. Floreano. Reducing Human Design and Increasing Adaptability in Evolutionary Robotics. In T. Gomi, editor, *Evolutionary Robotics*. AAI Books, Ontario, 1997.

[62] G. Frank, G. Hartmann ans A. Jahnke, and M. Schäfer. An Accelerator for Neural Networks with Pulse-Coded Model Neurons. *IEEE Transactions on Neural Networks, Special Issue on Pulse Coupled Neural Networks*, 10:527–539, May 1999.

[63] M. Frean. The Upstart Algorithm: A Method for Constructing and Training Feed-Forward Neural Networks. *Neural Computation*, 2:198–209, 1990.

[64] B. Fritzke. Unsupervised ontogenic networks. In *Handbook of Neural Computation*, pages C2.4:1–C2.4:16. Institute of Physics Publishing and Oxford University Press, 1997.

[65] W. Gerstner and J.L. van Hemmen. How to describe neural activity – spikes, rates, or assemblies? In J.D. Cowan, G. Tesauro, and J. Alspector, editors, *Advances in Neural Information Processing Systems 6*, pages 463–470. Morgan Kaufmann Publishers, 1994.

[66] C. Gilbert. Neural plasticity. In *The MIT Encyclopedia of the Cognitive Sciences*, pages 598–601. The MIT Press, 1999.

[67] B. Girau and A. Tisserand. On-line Arithmetic-Based Reprogrammable Hardware Implementation of Multilayer Perceptron Back-Propagation. In *Proceedings of MicroNeuro'96*, pages 168–175, 1996.

[68] M. Goeke, M. Sipper, D. Mange, A. Stauffer, E. Sanchez, and M. Tomassini. Online autonomous evolware. In T. Higuchi, M. Iwata, and L. Weixin, editors, *Evolvable Systems: From Biology to Hardware*, volume 1259 of *Lecture Notes in Computer Science*, pages 96–106. Springer-Verlag, 1997.

[69] R. Goldstone. Similarity. In *The MIT Encyclopedia of the Cognitive Sciences*, pages 763–765. The MIT Press, 1999.

[70] S. Grossberg. Adaptive pattern classification and universal recoding: I. Parallel development and coding of neural feature detectors. *Biological Cybernetics*, 21:117–158, 1976.

[71] S. Grossberg. Adaptive pattern classification and universal recoding: II. Parallel development and coding of neural feature detectors. *Biological Cybernetics*, 23:187–202, 1976.

[72] S. Grossberg. The Attentive Brain. Technical Report CAS/CNS-TR-95-012, Department of Cognitive and Neural Systems, Boston University, 1995. (also in American Scientist 83, 438–449, 1995).

[73] S. Grossberg. The Link between Brain Learning, Attention, and Conciousness. Technical Report CAS/CNS-TR-97-018, Department of Cognitive and Neural Systems, Boston University, June 1998. (also in Conciousness and Cognition, 8, 1, 1999).

[74] M. Gschwind, V. Salapura, and O. Maischberger. RANNSOM: A Reconfigurable Neural Network Architecture Based on Bit Stream Arithmetic. In *Proceedings of the Fourth International Conference on Microelectronics for Neural Networks and Fuzzy Systems*, Turin, Italy, September 1994.

[75] V. Gullapalli. A Comparison of Supervised and Reinforcement Learning Methods on a Reinforcement Learning Task. Computer and Information Science Department, University of Massachussetts, Amherst (unpublished paper), 1992.

[76] J.D. Hadley and B.L. Hutching. Design Methodologies for Partially Reconfigured Systems. In P. Athanas and K.L. Pocek, editors, *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*, pages 78–84, Los Alamitos, CA., April 1995. IEEE Computer Society Press.

[77] A. Hamilton, S. Churcher, P.J. Edwards, G.B. Jackson, A.F. Murray, and H.M. Reekie. Pulse stream VLSI circuits and systems: the epsilon neural network chipset. In *Proceedings of the Third International Conference on Microelectronics for Neural Networks*, volume 4, pages 395–405, December 1993.

[78] S. Hamilton and L. Garber. Deep Blue's Hardware-Software Synergy. *IEEE-Computer*, 1997.

[79] J. Hampton. Cencepts. In *The MIT Encyclopedia of the Cognitive Sciences*, pages 176–179. The MIT Press, 1999.

[80] S. Harnad, S.J. Hanson, and J. Lubin. Learned Categorical Perception in Neural Nets: Implications for Symbol Grounding. In V. Honavar and L. Uhr, editors, *Symbol Processors and Connectionist Network Models in Artificial Intelligence and Cognitive Modelling: Steps Toward Principled Integration*, pages 191–206. Academic Press, 1995.

[81] S. Hauck. The Roles of FPGA's in Reprogrammable Systems. *Proceedings of the IEEE*, 86(4):615–638, April 1998.

[82] J.R. Heath, P.J. Kuekes, G.S. Snider, and R.S. Williams. A Defect-Tolerant Computer Architecture: Opportunities for Nanotechnology. *Science*, 280:1716–1721, June 12 1998.

[83] D.O. Hebb. *The Organization of Behavior*. Wiley, New York, 1949.

[84] J. Hérault and C. Jutten. *Réseaux neuronaux et traitement du signal*. Traitement du signal. Hermes, Paris, second edition, 1994.

[85] K. Hertz, A. Krogh, and R. Palmer. *Introduction to the Theory of Neural Computation*, chapter 9, pages 217–243. Addison-Wesley, Redwood City,CA, 1991.

[86] J.H. Holland. *Adaptation in Natural and Artificial Systems*. University of Michigan Press, Ann Arbor, 1975. (Second edition by The MIT Press, Cambridge MA, 1992).

[87] J.H. Holland. *Hidden Order: How Adaptation Builds Complexity*. Addison Wesley, 1995.

[88] J.J. Hopfield. Neural network and physical systems with collective computational abilities. *Proceedings of the National Academy of Science*, 79:2554–2558, 1982.

[89] K. Hornik, M. Stinchcombe, and H. White. Multilayer feedforward networks are universal approximators. *Neural Networks*, 2:359–366, 1989.

[90] P. D. Hortensius, R. D. McLeod, W. Pries, D. M. Miller, and H. C. Card. Cellular Automata-Based Pseudorandom Number Generators for Built-In Self-Test. *IEEE Transactions on Computer-Aided Design*, 8(8):842–859, August 1989.

[91] P.D. Hortensius, R.D. McLeod, and H.C. Card. Parallel random number generation for VLSI systems using cellular automata. *IEEE Transactions on Computers*, 38(10):1466–1473, October 1989.

[92] F.-H. Hsu, T. Anantharaman, M. Campbell, and A. Nowatzyk. A Grandmaster Chess Machine. *Scientific American*, pages 18–23, October 1990.

[93] D.H. Hubel and T.N. Wiesel. Brain Mechanisms of Vision. *Scientific American*, 241(1), 1979.

[94] I-CUBE Inc. *I-CUBE. The FPID Family Data Sheet*, 2.0 edition, May 1994.

[95] IBM France, Component Development Laboratory. *ZISC 036 Neurons User's Manual*, May 15 1998.

[96] P. Ienne. Architectures for Neurocomputers: Review and Performance Evaluation. Technical Report TR. 93/21, Swiss Federal Institute of Technology-Lausanne, January 1993.

[97] P. Ienne. *Programmable VLSI Systolic Processors for Neural Network and Matrix Computations*. PhD thesis, Swiss Federal Institute of Technology-Lausanne, Lausanne, EPFL, 1996. Thesis 1525.

[98] P. Ienne, T. Cornu, and G. Kuhn. Special-purpose digital hardware for neural networks: An architectural survey. *Journal of VLSI Signal Processing*, pages 13(1):5–25, 1996.

[99] P. Ienne and G. Kuhn. Digital systems for neural networks. In P. Papamichalis and R. Kerwin, editors, *Digital Signal Processing Technology*, volume CR57 of *Critical Review Series*, pages 314–345. SPIE Optical Engineering, 1995.

[100] A. Jain, J. Mao, and K. Mohiuddin. Artificial neural networks: A tutorial. *IEEE Computer*, pages 31–44, March 1996.

[101] W. James. *The Principles of Psychology*. Dover, New York, 1890.

[102] M.I. Jordan and S. Russell. Computational Intelligence. In *The MIT Encyclopedia of the Cognitive Sciences*, pages lxxiii–xc. The MIT Press, 1999.

[103] K-Team SA, Lausanne, Switzerland. *Khepera User Manual*, verdion 4.06 edition, November 1995.

[104] L.P. Kaelbling, M.L. Littman, and A.W. Moore. Reinforcement Learning: A Survey. *Artificial Intelligence Research*, 4:237–285, 1996.

[105] R.E. Kalil. Synapse Formation in the Developing Brain. *Scientific American*, pages 38–45, December 1990.

[106] S. Kaski and T. Kohonen. Winner-Take-All Networks for Physiological Models of Competitive Learning. *Neural Networks*, 7(6/7):973–984, 1994.

[107] T. Kohonen. *Self-Organizing Maps*. Number 30 in Information Sciences. Springer, second edition, 1997.

[108] J.F. Kolen. *Exploring the Computational Capabilities of Recurrent Neural Networks*. PhD thesis, Ohio State University, 1994.

[109] K. Kollmann, K. Riemschneider, and H.C. Zeidler. On-chip Backpropagation training using parallel stochastic bit streams. In *Proceedings of the IEEE International Conference on Microelectronics for Neural Networks and Fuzzy Systems Microneuro'96*, pages 149–156, 1996.

[110] J.R. Koza. *Genetic Programming.* The MIT Press, Cambridge, Massachusetts, 1992.

[111] R.M. Kretchmar and C.W. Anderson. Comparison of CMACs and Radial Basis Functions for Local Function Approximators in Reinforcement Learning. In *Proceedings of the International Conference on Neural Networks, ICNN'97*, June 1997.

[112] B.J.A. Kröse and J.W.M van Dam. Adaptive state space quantization for reinforcement learning collision-free navigation. In *Proceedings of the 1992 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 1327–1332, 1992.

[113] L. Kuvayev and R. Sutton. Approximation in Model-Based Learning. In *Proceedings of the ICML'97 Workshop on Modelling in Reinforcement Learning*, Vanderbilt University, July 1997.

[114] R. A. Levinson. A self-learning, pattern-oriented Chess program. *ICCA Journal*, 12(4):207–215, 1989.

[115] C.-J. Lin and C.-T. Lin. Reinforcement Learning for An ART-Based Fuzzy Adaptive Learning Control Network. *IEEE Transactions on Neural Networks*, 7(3):709–731, 1996.

[116] J.L. Lin and T.M. Mitchell. Reinforcement learning with hidden states. In J-A. Meyer, H.L. Roitblat, and S.W. Wilson, editors, *From Animals to Animats: Proceedings of the Second International Conference on Simulation of Adaptive Behavior (SAB92)*, pages 281–290, 1992.

[117] C.S. Lindsay. Hardware neural networks. Lecture at Chalmers University in Göteborg, March 1998.

[118] C.S. Lindsey and T. Lindblad. Review of hardware neural networks: a user's perspective. In D. J. Amit et al. (Eds.), editor, *Third Workshop on Neural Networks: From Biology to high Energy Physics*, volume 6 of *International Journal of Neural Systems*. World Scientific, Singapure, 1995.

[119] M.L. Littman. Memoryless Policies: Theoretical Limitations and Practical Results. In D. Cliff, P. Husbands, J. Meyer, and S.W. Wilson, editors, *From Animals to Animats: Proceedings of the Third International Conference on Simulation of Adaptive Behavior (SAB94)*, pages 238–247, 1994.

[120] E. Littmann and H. Ritter. Adaptive Color Segmentation- A comparison of neural and statistical methods. In *IEEE Transactions on Neural Networks*, pages 175–185, January 1996.

[121] J. Loch and S. Singh. Using Eligibilities traces to find the best memoryless policy in partially observable Markov decission processes. In *Proceedings of the International Conference on Machine Learning ICML'98*. Morgan Kaufmann, 1998.

[122] Y. López-De-Meneses and O. Michel. Vision Sensors on the Webots Simulator. In *Proceedings of Virtual Worlds'98*, pages 264–273, Paris, France, 1-3 July 1998.

[123] H.H. Lund, B. Webb, and J. Hallam. A Robot Attracted to the Cricket Species Gryllus bimaculatus. In *Proceedings of 4th European Conference on Artificial Life 1997*. The MIT Press, 1997.

[124] W. Maass. Vapnik-chervonenkis dimension of neural networks. In M.A. Arbib, editor, *Handbook of Brain Theory and Neural Networks*, pages 1000–1003. MIT Press, 1995.

[125] W. Maass. Networks of Spiking neurons: The third generation of neural network models. In *Proceedings of the 7th Australian Conference on Neural Networks 1996*, pages 1–10, 1996.

[126] J. MacCarthy. AI as Sport. *Science*, 276(5318):1518–1519, June 6 1996.

[127] S. Mahadevan and J. Connell. Automatic Programming of Behavior-based Robots using Reinforcement Learning. *Artificial Intelligence*, 55(2-3):311–365, June 1992.

[128] H.A. Mallot. Layered computaion in neural networks. In M.A. Arbib, editor, *Handbook of Brain Theory and Neural Networks*, pages 513–516. MIT Press, 1995.

[129] D. Mange, E. Sanchez, A. Stauffer, G. Tempesti, P. Marchal, and C. Piguet. Embryonics: A New Methodology for Designing Field-Programmable Gate Arrays with Self-Repair and Self-Replicating Properties. *IEEE Transactions on VLSI Systems*, 6(3):387–399, September 1998.

[130] D. Mange and M. Tomassini, editors. *Bio-Inspired Computing Machines: Toward Novel Computational Machines*. Presses Polytechniques et Universitaires Romandes, Lausanne, Switzerland, 1998.

[131] P. Marchal and P. Nussbaum. Demultiplexer-Based Cells. In D. Mange and M. Tomassini, editors, *Bio-Inspired Computing Machines: Toward Novel Computational Machines*, pages 167–182. Presses Polytechniques et Universitaires Romandes, Lausanne, Switzerland, 1998.

[132] M. Marchesi, G. Orlandi, F. Piazza, and A. Uncini. Fast Neural Networks Without Multipliers. *IEEE Transactions on Neural Networks*, 4(1):53–62, January 1993.

[133] W. McCulloch and W. Pitts. A logical calculus of the ideas immanent in neural nets. *Bulletin of Math. Biophys.*, pages 115–137, 1943.

[134] C. Mead. *Analog VLSI and Neural Sytems*. Addison-Wesley, May 1989.

[135] D.L. Medin and C.M. Aguilar. Categorization. In *The MIT Encyclopedia of the Cognitive Sciences*, pages 104–106. The MIT Press, 1999.

[136] D. Michie and R.A. Chambers. BOXES: An experiment in adaptive control. In E.Dale and D. Michie, editors, *Machine Intelligence 2*, pages 137–152. Edinburgh Oliver and Boyd, 1968.

[137] J.R. Millan. Rapid, Safe, and Incremental Learning of Navigation Strategies. *IEEE Transactions on Systems, Man and Cybernetics (Part B)*, 26(3):408–420, June 1996.

[138] W.T. Miller-III, R.S. Sutton, and P.J. Werbos, editors. *Neural Networks for Control*. The MIT Press, 1995.

[139] F. Mondada, E. Franzi, and P. Ienne. Mobile robot miniaturization: A tool for investigating in control algorithms. In *Proceedings of the Third International Symposium on Experimental Robotics*, Kyoto, Japan, 1993.

[140] J. Moody and C. Darken. Learning with localized receptive fields. In D. Touretsky, G. Hinton, and T. Sejnowski, editors, *Proceedings of the 1988 Connectionist Models*, pages 133–143. Summer School Carnegie Mellun University, Morgan Kaufmann, June 17-26 1988.

[141] J. Moody and C. Darken. Fast Learning in Networks of Locally-Tunned Processing Units. *Neural Computation*, 1:281–294, 1989.

[142] A.W. Moore and C.G. Atkeson. Memory-Based Reinforcement Learning: Efficient Computation with Prioritized Sweeping. In S.J. Hanson, J.D. Cowan, and C. Lee Giles, editors, *Advances in Neural Information Processing Systems*, volume 5, pages 263–270, San Mateo, CA, 1993. Morgan Kaufmann.

[143] B. Moore. ART 1 and Pattern Clustering. In D. Touretsky, G. Hinton, and T. Sejnowski, editors, *Proceedings of the 1988 Connectionist Models*, pages 174–185. Summer School Carnegie Mellun University, Morgan Kaufmann, June 17-26 1988.

[144] J.M. Moreno, J. Madrenas, S. San Anselmo, F. Castillo, and J. Cabestany. Digital Hardware Implementation of ROI Incremental Algorithms. In J. Mira and F. Sandoval, editors, *From natural to Artificial Neural Computation*, volume 930 of *Lecture Notes in Computer Science*, pages 761–770. Springer Verlag, 1995.

[145] J.M. Moreno, J. Madrenas, J. Faura, J. Cabestany, and J.M. Insenser. Feasible Evolutionary and Self-Repairing Hardware by Means of the Dynamic Reconfiguration Capabilities of the FIPSOC Devices. In M. Sipper, D. Mange, and A. Pérez-Uribe, editors, *Evolvable Systems: from Biology to Hardware*, volume 1478 of *Lecture Notes in Computer Science*, pages 345–355. Springer-Verlag, 1998.

[146] J.M. Moreno-Aróstegui. *VLSI Architecture for Evolutive Neural Models*. PhD thesis, Universitat Polytècnica de Catalunya, December 1994.

[147] E. Mosanya, M. Goeke, J. Linder, J.-Y. Perrier, F. Rampogna, and E. Sanchez. A platform for Co-design and Co-synthesis based on FPGA. In *Proceedings of the 7th IEEE International Workshop on Rapid System Prototyping*, pages 11–16, 1996.

[148] S. Nolfi, D. Parisi, and J. L. Elman. Learning and evolution in neural networks. *Adaptive Behavior*, 3(1):5–28, 1994.

[149] Y. Ohta. *Knowledge-Based Interpretation of Outdoor Natural Color Scenes*. Pitman Advanced Publishing Program, 1985.

[150] D.K. Olson. Learning to play games from experience: An application of artificial neural networks and temporal difference learning. Master's thesis, Pacific Lutheran University, Washington, 1993.

[151] J. Peng and R.J. Williams. Efficient Learning and Planning Within the Dyna Framework. *Adaptive Behavior*, 1(4):437–454, 1993.

[152] A. Pérez-U and E. Sanchez. The FAST Architecture: A Neural Network with Flexible Adaptable-Size Topology. In *Proceedings of the Fourth International Conference on Microelectronics for Neural Networks and Fuzzy Systems Microneuro'96*, pages 337–340, Lausanne, Switzerland, February 1996. IEEE.

[153] A. Pérez-Uribe. Artificial Neural Networks: Algorithms and Hardware Implementation. In D. Mange and M. Tomassini, editors, *Bio-Inspired Computing Machines: Toward Novel Computational Machines*, pages 289–316. Presses Polytechniques et Universitaires Romandes, Lausanne, Switzerland, 1998.

[154] A. Pérez-Uribe and E. Sanchez. FPGA Implementation of an Adaptable-Size Neural Network. In C. von der Malsburg, W. von Seelen, J. C. Vorbrüggen, and B. Sendhoff, editors, *Proceedings of the International Conference on Artificial Neural Networks (ICANN96)*, volume 1112 of *Lecture Notes in Computer Science*, pages 383–388. Springer-Verlag, Heidelberg, 1996.

[155] A. Pérez-Uribe and E. Sanchez. Neural Networks Structure Optimization through On-line Hardware Evolution. In *Proceedings of the World Congress on Neural Networks (WCNN96)*, pages 1041–1044. INNS (International Neural Networks Society) Press, 1996.

[156] A. Pérez-Uribe and E. Sanchez. Speeding-Up Adaptive Heuristic Critic Learning with FPGA-based Unsupervised Clustering. In *Proceedings of the International Conference on Evolutionary Computation ICEC97*, pages 685–689, IEEE Press, April 1997.

[157] A. Pérez-Uribe and E. Sanchez. Structure-Adaptable Neurocontrollers: A Hardware-Friendly Approach. In Roberto Moreno-Díaz José Mira and Joan Cabestany, editors, *Biological and Artificial Computation: From Neuroscience to technology*, pages 1251–1259, Lecture Notes in Computer Science 1240, Springer Verlag, 1997.

[158] A. Pérez-Uribe and E. Sanchez. Blackjack as a Test Bed for Learning Strategies in Neural Networks. In *Proceedings of the IEEE International Joint Conference on Neural Networks IJCNN'98*, volume 3, pages 2022–2027, Anchorage, May 4-9 1998.

[159] A. Pérez-Uribe and E. Sanchez. A Digital Artificial Brain Architecture for Mobile Autonomous Robots. In M. Sugisaka and H. Tanaka, editors, *Proceedings of the Fourth International Symposium on Artificial Life and Robotics AROB'99*, pages 240–243, Oita, Japan, 1999.

[160] A. Pérez-Uribe and E. Sanchez. Structure Adaptation in Artificial Neural Networks through Adaptive Clustering and through Growth in State Space. In J. Mira and J. V. Sánchez-Andrés, editors, *Foundations and Tools for Neural Modeling*, volume I of *Lecture Notes in Computer Science 1606*, pages 556–565. Springer Verlag, 1999.

[161] J. Platt. A Resource-Allocating Network for Function Interpolation. *Neural Computation*, 3:213–225, 1991.

[162] T. Poggio. Vision and Learning. In *The MIT Encyclopedia of the Cognitive Sciences*, pages 863–864. The MIT Press, 1999.

[163] T. Poggio and F. Girosi. Regularization Algorithms for Learning That Are Equivalent to Multilayer Networks. *Science*, 1990.

[164] S. R. Quartz and T. J. Sejnowski. The neural basis of cognitive development: A constructivism manifesto. *Behavioral and Brain Sciences*, 20(4):537+, December 1997.

[165] P. Rakic, J-B. Bourgeois, and P.S. Goldman-Rakic. Synaptic development of the cerebral cortex: implications for learning, memory, and mental illness. In J. van Pelt, M.A. Corner, H.B.M. Uylings, and F.H. Lopes da Silva, editors, *Progress in Brain Research*, volume 102, pages 227–243. Elsevier Science, 1994.

[166] R. Reed. Pruning Algorithms-A Survey. *IEEE Transactions on Neural Networks*, 4(5):740–747, September 1993.

[167] D.L. Reilly, L.N. Cooper, and C. Elbaum. A Neural Model for Category Learning. *Biological Cybernetics*, pages 35–41, 1982.

[168] L.M. Reyneri. Weighted Radial Basis Functions for Improved Pattern Recognition and Signal Processing. *Neural Processing Letters*, 2(3):2–6, 1995.

[169] M.B. Ring. *Continual learning in reinforcement environments.* PhD thesis, The University of Texas at Austin, August 1994.

[170] B.D. Ripley. *Pattern Recognition and Neural Networks.* Cambridge University Press, 1996.

[171] F. Rosenblatt. *Principles of Neurodynamics:Perceptrons and the theory of brain mechanics*. Spartan Books, Washington D.C., 1962.

[172] A. Roy. Brain's internal mechanisms - a new paradigm. In *Proceedings of the International Joint Conference on Neural Networks (IJCNN'99)*, page (to appear), Washington D.C., 1999.

[173] A. Roy and R. Miranda. Fuzzy Logic, Neural Networks and Brain-like Learning. In *Proceedings of the International Conference on Neural Networks (ICNN'97)*, volume 1, pages 522–527, Houston, 1997.

[174] D.E. Rumelhart, G.E. Hinton, and R.J. Williams. Learning internal represenations by error propagation. In D.E. Rumelhart, J.L. McClelland, and PDP Research Group, editors, *Parallel Distributed Processing: Explorations in the Microstructure of Coginition*, volume 1, pages 318–362. The MIT Press, Cambridge, MA, 1986.

[175] G. Rummery and M. Niranjan. On-line q-learning using connectionist systems. Technical Report CUED/F-INFENG/TR 166, Cambridge University Engineering Department, 1994.

[176] A.L. Samuel. Some studies in Machine Learning Using the Game of Checkers. *IBM Journal*, pages 211–229, July 1959.

[177] E. Sanchez. Field Programmable Gate Array (FPGA) circuits. In Springer Verlag, editor, *Towards Evolvable Hardware*, pages 1–18, 1996.

[178] E. Sanchez. An introduction to digital systems. In D. Mange and M. Tomassini, editors, *Bio-Inspired Computing Machines: Toward Novel Computational Machines*, pages 13–47. Presses Polytechniques et Universitaires Romandes, Lausanne, Switzerland, 1998.

[179] E. Sanchez, M. Sipper, J.-O. Haenni, J.-L. Beuchat, A. Stauffer, and A. Pérez-Uribe. Static and Dynamic Configurable Systems. *IEEE Transactions on Computers*, 48(6):556–564, June 1999.

[180] S. Schaal and C.G. Atkeson. Constructive Incremental Learning From Only Local Information. *Neural Computation*, 10(8):2047–2084, November 1998.

[181] C. Scheier. Incremental Category Learning in a Real World Artifact Using Growing Dynamic Cell Structures. In M. Verleysen, editor, *Proceedings of the 4th European Symposium on Artificial Neural Networks ESANN'96*, pages 117–122, Bruges, 1996.

[182] G. Schram, B.J.A. Kröse, R. Babuska, and A.J. Krijgsman. Neurocontrol by Reinforcement Learning. *Journal A (Journal on Automatic Control)*, 37(3):59–64, 1996.

[183] N. N. Schraudolph, P. Dayan, and T.J. Sejnowski. Temporal Difference Learning of Position Evaluation in the Game of Go. In J. D. Cowan, G. Tesauro, and J. Alspector, editors, *Advances in Neural Information Processing Systems*, volume 6, pages 817–824. Morgan Kaufmann Publishers, 1994.

[184] W. Schultz, P. Dayan, and P. Read Montague. A Neural Substrate of Prediction and Reward. *Science*, 275:1593–1599, 14 March 1997.

[185] A. Schütz. Neuroanatomy in a computational perspective. In M.A. Arbib, editor, *Handbook of Brain Theory and Neural Networks*, pages 622–626. MIT Press, 1995.

[186] J. Searle. Minds, Brains, and Programs. *Behavioral and Brain Sciences*, 3:417–424, 1980.

[187] J. Searle. Is the Brain's Mind a Computer Program? *Scientific American*, 262:26–31, 1990.

[188] T. Serrano-Gotarredonda, B. Linares-Barranco, and A.G. Andreou. *Adaptive Resonance Theory Microchips*, volume 456 of *Series in engineering and computer science*. Kluwer Academic, Boston, August 1998.

[189] C.E. Shannon. A Chess-Playing Machine. *Scientific American*, February 1950. (also in Claude Elwood Shannon. Collected Papers, J.A. Sloane and A.D. Wyner (Eds.), IEEE Press, pp. 657-666, 1992).

[190] C.E. Shannon. Presentation of a Maze-Solving Machine. In *Transactions of the 8th Cybernetics Conference*. Josiah Macy Jr. Foundation, 1952. (also in Claude Elwood Shannon. Collected Papers, J.A. Sloane and A.D. Wyner (Eds.), IEEE Press, pp. 681-687, 1992).

[191] C.J. Shatz. The Developing Brain. *Scientific American*, pages 35–41, September 1992.

[192] J. Shawe-Taylor, P. Jeavons, and M. van Daalen. Probabilistic Bit Stream Neural Chip: Theory. *Connection Science*, 3(3):317–328, 1991.

[193] M. Sipper, E. Sanchez, D. Mange, M. Tomassini, A. Pérez-Uribe, and A. Stauffer. A Phylogenetic, Ontogenetic, and Epigenetic View of Bio-Inspired Hardware Systems. *IEEE Transactions on Evolutionary Computation*, 1(1):83–97, April 1997.

[194] J. Sirosh and R. Miikkulainen. Cooperative self-organization of afferent and lateral connections in cortical maps. *Biological Cybernetics*, 71(1):66–78, 1994.

[195] B.F. Skinner. *The Behavior of Organisms*. Appleton-Century-Crofts, New York, 1938.

[196] D.F. Specht. Probabilistic neural networks. *Neural Networks*, 3:109–118, 1990.

[197] R.S. Sutton. Learning to predict by the methods of Temporal Differences. *Machine Learning*, 3:9–44, 1988.

[198] R.S. Sutton. Integrated architectures for Learning, Planning, and Reacting based on approximating Dynamic Programming. In Morgan Kaufmann, editor, *Proceedings of the Seventh International Conference on Machine Learning*, pages 216–224, 1990.

[199] R.S. Sutton. Personal communication, February 1998.

[200] R.S. Sutton and A.G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, 1998.

[201] R.S. Sutton, D. McAllester, S. Singh, and Y. Mansour. Policy Gradient Methods for Reinforcement Learning with Function Approximation. 1999 (Submitted).

[202] J. Tani. Model-Based Learning for Mobile Robot Navigation from the Dynamical Systems Perspective. *IEEE Transactions on Systems, Man and Cybernetics (Part B)*, 26(3):421–436, June 1996.

[203] G. Tempesti. *A Self-Repairing Multiplexer-Based FPGA Inspired by Biological Processes*. PhD thesis, Swiss Federal Institute of Technology-Lausanne, Lausanne, EPFL, 1998. Thesis 1827.

[204] G. Tesauro. TD-Gammon, a Self-Teaching Backgammon Program, Achieves Master-Level Play. *Neural Computation*, 6(2):215–219, 1994.

[205] G. Tesauro. Temporal Difference Learning and TD-Gammon. *Communications of the ACM*, 38(3):58–68, March 1995.

[206] C.K. Tham and R.W. Prager. Reinforcement Learning for Multi-Linked Manipulator Control. Technical Report CUED/F-INFENG/TR 104, Cambridge University, Engineering Department, June 24 1992.

[207] A. Thompson. Artificial Evolution in the Physical World. In T. Gomi, editor, *Evolutionary Robotics: From Intelligent Robots to Artificial Life (ER'97)*, pages 101–125. AAI Books, 1997.

[208] E.L. Thorndike. *Animal Intelligence: Experimental Studies*. McMillan, 1911.

[209] S. Thrun. The role of exploration in learning control. In D.A. White and D.A. Sofge, editors, *Handbook of Intelligent Control: Neural, Fuzzy and Adaptive Approaches*, pages 527–559. Van Nostrand Reinhold, New York, 1992.

[210] S. Thrun. Exploration in Active Learning. In M.A. Arbib, editor, *Handbook of Brain Theory and Neural Networks*, pages 381–384. The MIT Press, 1995.

[211] N. Tredennick. Microprocessor-based computers. *IEEE Computer: 50 years of computing*, pages 27–37, October 1996.

[212] S. M. Trimberger. *Field-Programmable Gate Array Technology*. Kluwer Academic Publishers, Boston, 1994.

[213] A.M. Turing. Intelligent Machinery. Report, National Physics Laboratory, 1948. (also in Mechanical Intelligence, Collected Works of A.M. Turing, North Holland, 1992).

[214] A.M. Turing. Computing machinery and intelligence. *Mind*, LIX(360):433–460, 1950.

[215] P. Turney. Myths and Legends of the Baldwin Effect. In *Proceedings of the 13th International Conference on Machine Learning (ICML-96)*, pages 135–142, 1996.

[216] J. Vauclair. *L'intelligence de l'animal*. Éditions du Seuil, Paris, October 1995.

[217] J. Villasenor and B. Hutchings. The Flexibility of Configurable Computing. *IEEE Signal Processing Magazine*, 15(5):67–84, September 1998.

[218] J. Villasenor and W.H. Mangione-Smith. Configurable Computing. *Scientific American*, 276(6):54–59, June 1997.

[219] E. Vittoz. Analog VLSI Signal Processing: Why, Where and How? *Journal of VLSI Signal Processing*, 8:27–44, July 1994.

[220] C. von der Malsburg. Self-organization of orientation sensitive cells in the striate cortex. *Kybernetik*, 14:85–100, 1973.

[221] J. von Neumann. *The Computer and the Brain*. Yale University Press, 1958.

[222] C.J.C.H. Watkins and P. Dayan. Q-Learning. *Machine Learning*, 8:279–292, 1992.

[223] A. Watson. Why Can't a Computer Be More Like a Brain ? *Science*, 277(5334):1934–1936, September 1997.

[224] P.J. Werbos. *The Roots of Backpropagation: From ordered derivatives to Neural Neworks and Political Forecasting*. John Wiley and Sons, New York, 1994.

[225] P.J. Werbos. Bckpropagation: Basics and new developments. In M.A. Arbib, editor, *Handbook of Brain Theory and Neural Networks*, pages 134–139. MIT Press, 1995.

[226] D.A. White and D.A. Sofge, editors. *Handbook of Intelligent Control*. Van Nostrand Reinhold, New York, 1992.

[227] S.D. Whitehead and D.H. Ballard. Learning to Perceive and Act by Trial. *Machine Learning*, 7(1):45–83, 1991.

[228] B. Widrow, N. Gupta, and S. Maitra. Punish/Reward: Learning with a Critic in Adaptive Threshold Systems. *IEEE Transactions on Systems, Man and Cybernetics*, 3(5):455–465, September 1973.

[229] B. Widrow and M. Lehr. Perceptrons, Adalines, and Backpropagation. In M.A. Arbib, editor, *Handbook of Brain Theory and Neural Networks*, pages 719–724. MIT Press, 1995.

[230] B. Widrow and F. Smith. Pattern-recognizing Control Systems. In *Proceedings of the 1963 Computer and Information Sciences (COINS) Symposium*, pages 288–317, Washington D.C, 1964.

[231] N. Wiener. *God & Golem, Inc. A comment on Certain Points where Cybernetic Impinges on Religion*, page 14. The MIT Press, 1964.

[232] R.J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 8:229–256, 1992.

[233] S.W. Wilson. ZCS: a zeroth level classifier system. *Evolutionary Computation*, 2(1):1–18, 1994.

[234] S.W. Wilson. Classifier fitness based on accuracy. *Evolutionary Computation*, 3(2):149–175, 1995.

[235] S.W. Wilson. Explore/Exploit Strategies in Autonomy. In *From Animals to Animats 4: Proceedings of the Fourth International Conference on Simulation of Adaptive Behavior*, pages 325–332, Cambridge, MA, 1996. The MIT Press.

[236] S. Wolfram. Statistical mechanics of cellular automata. *Review of Modern Physics*, 55:601–644, 1983.

[237] Xilinx. *The XC4000 Data Book*. Xilinx,Inc, San Jose, 1991.

[238] Xilinx,Inc. *XC6200 Field Programmable Gate Arrays*, April 1997.

[239] Xilinx,Inc. *XC4000XV Family Field Programmable Gate Arrays*, May 1998.

[240] X. Yao. Evolutionary artificial neural networks. *International Journal of Neural Systems*, 4(3):203–222, 1993.

[241] M. Zeidenberg. *Neural Networks in Artificial Intelligence*, chapter Issues in Neural Network Modeling. Ellis Horwood, 1990.

[242] J. Zhao, J. ShaweTaylor, and M van Daalen. Learning in stochastic bit stream neural networks. *Neural Networks*, 9(6):991–998, 1996.

[243] S. Zrehen. *Elements of Brain Design for Autonomous Agents*. PhD thesis, Swiss Federal Institute of Technology-Lausanne, Lausanne, EPFL, 1995. Thesis 1373.

Curriculum Vitæ

"Ma bibliothèque, c'est moi."

-J. Holland

Andrés Pérez-Uribe received a diploma from the Universidad del Valle, Cali, Colombia, in 1993. From 1994 to 1996, he held a Swiss government fellowship and is currently a PhD candidate in the Department of Computer Science, Swiss Federal Institute of Technology, Lausanne. Since 1994, he has been with the Logic Systems Laboratory at the Swiss Federal Institute of Technology, working on the digital implementation of neural networks with adaptable topologies, in collaboration with the Centre Suisse d'Electronique et de Microtechnique SA (CSEM). His research interests include artificial neural networks, field-programmable devices, evolutionary techniques, and complex and bio-inspired systems. He was a member of the steering committee and secretary of the Second International Conference on Evolvable Systems: From Biology to Hardware (ICES'98) held in Lausanne, Switzerland, in September 1998.